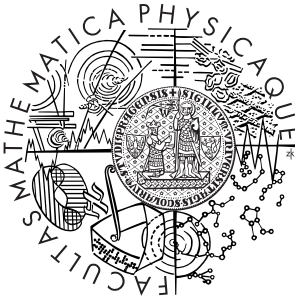
Charles University in Prague
Faculty of Mathematics and Physics

DOCTORAL THESIS



Viliam Holub

Fighting the State Explosion Problem
in Component Protocols

Department of Software Engineering
Advisor: Prof. František Plášil

Abstract

Title: Fighting the state explosion problem in component protocols
Author: Viliam Holub
email: holub@dsrg.mff.cuni.cz
phone: +420 2 2191 4235
Department: Department of Software Engineering
Faculty of Mathematics and Physics
Charles University in Prague, Czech Republic
Advisor: Prof. Ing. František Plášil, DrSc.
email: plasil@dsrg.mff.cuni.cz
phone: +420 2 2191 4266
Mailing address (both Author and Advisor):
Department of Software Engineering, Charles University in Prague
Malostranské nám. 25
118 00 Prague, Czech Republic
WWW: http://dsrg.mff.cuni.cz/

Abstract:

In complex software component systems, it is desirable to verify the correctness of the composition before deployment. To achieve a trustworthy composition, the behavior of components is formally described and the composition is verified against communication errors. Unfortunately, the number of states of a model tends to grow exponentially with the size of the model's description — the state explosion problem. Because the exhaustive verification has to visit all the states of the model, the verification leads to unacceptable space and time requirements.

In this thesis, we present several approaches to cope with the state explosion problem in behavior protocols. First, we reduce a size of the specification by enhancing the specification language by exceptions and, additionally, we reduce the specification by symbolic manipulations with respect to composition. Then, we present a novel approach to distributed verification, which involves external storage devices. Finally, we reduce the number of states, which have to be traversed by identifying representatives in the state space.

Keywords: symbolic optimizations and manipulations, parallel and distributed model verification, slicing, partial-order reduction

Acknowledgments

I am deeply grateful to my supervisor, Frantisek Plasil, for his support over my study and research. Also, he is a co-author of two included papers. I appreciate the help of Petr Tuma, a co-author of an included paper, with peer-reviewing my papers and his valuable suggestions. My work is often evaluated on real-life specifications created by Pavel Jezek, Jan Kofron, Ondrej Sery, and Tomas Poch and is also based on many internal discussions among other members of the Distributed systems research group, namely Petr Hnetynka, Jiri Adamek, Martin Mach, Tomas Kalibera, and Lubomir Bulej.

Last but not least, many thanks goes to my parents and Martina for their support and patience over my study.

Though my doctoral study, I was partially supported by the Czech Science Foundation (GACR) project 201/05/H014, the Grant Agency of the Czech Republic project 201/06/0770, and the Czech Academy of Sciences project 1ET400300504.

Contents

Chapter 1

Introduction

In recent years, industrial development of large software systems tends to use software components. While "software component" is a piece of software with well-defined communication interfaces [67], a software system is composed of several software components bound via interfaces. The idea of software components is inspired by a successful use of circuit elements in hardware manufacturing. Analogous to circuit elements are software components, connectors to interfaces, and wires to bindings.

It is common to distinguish interfaces required and provided, together with their static typing information. However, a simple type match is insufficient to guarantee the correctness of system composition, similarly as a signal voltage level does not guarantee correct assembly. Therefore, formal methods of describing software component's behavior take place.

Formal methods for expressing behavior over component interfaces require solid mathematical foundations. The developer of a software component is expected to formally describe its behavior. Then, the system specification is created by a composition of software component's behavior specification with respect to system architecture. Here, because of rapid system development, formal methods must allow automatic verification of the system specification; a proof of correctness in particular. Such a verification tool is often based on systematic traversing of the state space determined by the system specification.

1.1 Problem Statement

We can identify several problems in the application of formal methods to software component.

The size of the state space, which has to be traversed, tends to grow expo-

nentially with the size of the specification, particularly by employed parallel activities. The effect of model growing with respect to the specification size is well know in formal methods and is usually referred to as the *state explosion problem*. The state explosion problem is long-standing and one of the main obstacles in practical use of formal methods in software engineering. There are two reasons why. First, the required time to traverse the whole state space becomes unacceptable. Second, because most verification tools have to store generated states to avoid unnecessary re-exploring, memory requirements are proportional to the size of the model and set a barrier to the verifiable system size.

Additionally, a long verification runtime, required to traverse a large state space, increases the demand on reliability of the computational environment; often a failure of a single computational node leads to a termination of the whole verification and a lost of computational time (verification crashes). Since the most powerful environment relatively easy available in practice — the network of workstations — is also the least reliable, the verification tools face to the problem of *failure recovery*.

When applying formal methods to *already existing* software systems, software designers cope with the problem of *insufficient expressive power* of the specification language. While the specification language is suitable in the bottom-up design, problems raise in the opposite direction: Unbounded constructs used in the software do not fit to the formal description and have to be emulated in an overly complex manner or cannot be captured formally at all.

1.2 Goals of the Thesis

The goals of the thesis are to mitigate the problems sketched in the previous section while being partially focused on the Behavior protocols [9] formal language. In particular to:

1. Reduce the size of the specification with respect to the composition via symbolic manipulations and enhancing the specification language by constructs which had to be emulated overly complex otherwise.

2. Reduce time of the verification by aggregating available resources in a network of non-homogeneous, non-dedicated computers and to investigate options of failure recovery.

3. Identify possibilities for partial-order reductions in order to reduce the number of states which have to be visited during the verification.

1.3 Organization of the Thesis

The thesis is organized in eleven chapters as a collection of published papers, which represent my key contributions, along with chapters devoted to background, discussion, and related work.

In Chapter 2, we provide an overall introduction to the SOFA and Fractal component models, basic terminology, syntax and semantics of Behavior protocols, composition operators, and composition errors.

Then, we present related work in Chapter 3, particularly on process algebras, static and dynamic program slicing, approaches and problems of parallel and distributed model checking, and a use of high-latency mass storage devices in model checking.

In Chapter 4, we provide an overview and a brief discussion on the contribution of the papers from Chapters 5-10 to show how they address the goals stated in Sect. 1.2.

Chapter 5 presents an enhancement of Behavior protocols with explicit exceptions and our experiences with applying exceptions in formal description of a real-life, non-trivial project SPEEDO. The content of the chapter has been published as Exceptions in Component Interaction Protocols - necessity, co-authored with František Plášil, published in Architecting Systems with Trustworthy Components: International Seminar in LNCS [4].

Chapter 6 presents a method of formal specification reduction by symbolic manipulations. A set of reduction rules is described, together with discussion about their correctness with respect to composition. The paper has been accepter fro publication as Reducing Component Systems' Behavior Specification, co-authored with František Plášil, in the proceedings of the XXVI International Conference of the Chilean Computer Science Society (SCCC 2007) published by IEEE Computer Society [3]. Effects of reductions on an example formal specification are presented as well.

Chapter 7 describes our basic position on distributed model verification of generalized interaction protocols. The content of the chapter has been presented as On Distributed Verification of Generalized Interaction Models of Software Components at the 20^{th} European Conference on Object-Oriented Programming (ECOOP 2006) Doctoral Symposium [5].

Chapter 8 discusses problems of traditional approaches to distributed model verification, describes a new method based on a streamed state space ordering, and presents internals of the experimental implementation. Experimental results obtained with the method on large state spaces are provided as well, including the results on multi-processor systems and clusters. The content of the chapter has been published as Streaming State Space: A Method of Distributed Model Verification, co-authored with Petr Tůma, in the pro-

ceedings of the 1^{st} IEEE & IFIP International Symposium on Theoretical Aspects of Software Engineering (TASE 2007) published by IEEE Computer Society Press [2].

Chapter 9 presents an on-the-fly algorithm of identifying representatives in interfering automata. Experimental results on traditional and real-life examples are provided as well. The content of the chapter has been published as Identifying Representatives for Interfering Automata in the proceedings of 1^{st} Digital Communications and Computer Applications (DCCA 2007) published by the Jordan University of Science and Technology [1].

Chapter 10 describes an implementation of the log-structured file system developed in the scope of a student software project at Charles University. The content of the chapter has been published as Implementation of a Linux Log-Structured File System with a Garbage Collector, co-authored with Martin Jambor, Tomáš Hrubý, Jan Tauš, and Kuba Krchák and published in the Operating Systems Review special issue by ACM Press [8].

Finally, Chapter 11 draws conclusion and future work.

Most of the presented materials are freely accessible on the official Distributed Systems Research Group's web pages http://dsrg.mff.cuni.cz/ and on the web pages of the Object Web Consortium http://www.objectweb.org/.

Chapter 2

Background

2.1 SOFA

We illustrate the basic principles of the SOFA (SOFtware Aplliances) component model [44]. For elaborated description, we refer the reader to papers [11, 17, 42, 77].

In short, SOFA allows designing the application using Architecture Description Language, formally specify the behavior of software components, automatically generate connectors between software components, and dynamically update and distribute the application.

Applications are constructed of hierarchically composed software components (Fig. 2.1)). Two types of components are distinguished. Directly implemented components are called *primitive*, while those constructed by a combination of another components are *composite*.

Components communicate via *required* or *provided* interfaces. Implementation internals are hided to the user (a black-box view), so the published interfaces are the only connection between components. Bindings among

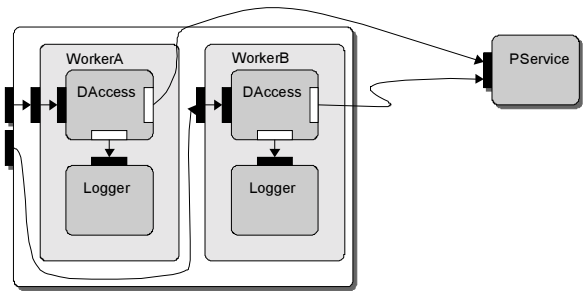


Figure 2.1: Example of SOFA components [48]

components are performed via connectors [17] which are first-class entities like components. Typically, bindings connect required with provided interfaces. Additionally, in composite components, bindings delegates between provided interfaces of a component and required interfaces of its subcomponent.

Recently, an enhanced version of SOFA 2.0 [45, 46, 48] has been developed. Along other features, it allows to define a component model via a meta-model, to reconfigure the architecture dynamically, to communicate among interfaces via connectors in practically any style, and to define and allow to use external services.

2.2 Fractal

The Fractal component model [39, 40] builds on hierarchically nested components (examples from the SPEEDO project [47], Fig. 2.2, 2.3). The Fractal specification [41] defines four levels of conformance. Each level specifies features, which must be provided to satisfy the specification. On level 0, there are no requirements and thus almost all the classes ale allowed components. On level 1, the component should provide the basic information about interfaces (component introspection). On level 2, the component should provide interface introspection, including name and type. Finally, on level 3, the component should provide typing and subtyping information.

Next to traditional required and provided interfaces, Fractal introduces several other types of interfaces. A role of *controller* interfaces is to manage the component. Controllers allow to stop or start a component (life cycle) and change the internal structure of composite components. *Optional* interfaces are not required to be connected. *Multiple* interfaces specify an array of interfaces.

Next to traditional (primitive) bindings, Fractal allows composite bindings. A composite binding allows interconnecting an arbitrary number of interfaces even with different types. It is represented by *binding components* referred to as connectors which are implemented as ordinary components. A unique feature among component models is a concept of *shared* components. A shared component is a part of several composite components.

Currently, there are several implementations that conform to the Fractal specification (not necessarily to the level 3), including SOFA. A reference implementation in the Java programming language is the open source Julia framework [43]. Specification of components in Julia has been enhanced by Behavior protocols as a part of the Component reliability extension for Fractal Component Model project [12].

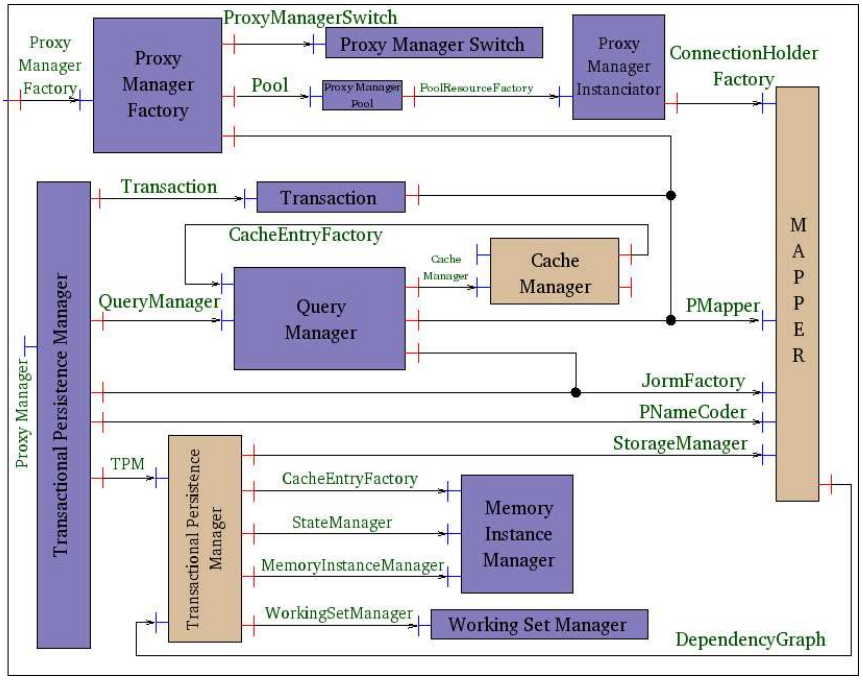


Figure 2.2: Componet model of the SPEEDO project [47]

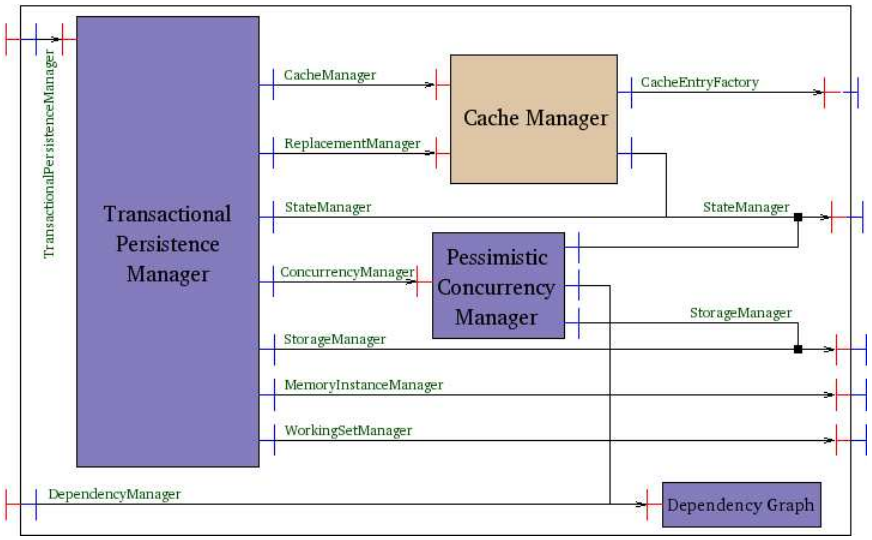


Figure 2.3: Composite component Transactional Persistence Manager from Fig. 2.2 [47]

An interesting achievent is the Fractive [70] implementation which uses the ProActive middleware [71]. ProActive is an implementation of distributed objects with asynchronous method calls realized via *future references* — references to return values which have not been obtained yet.

2.3 Behavior Protocols

Behavior protocols [9] have been developed as a specific process algebra describing behavior of software components. Although originally intended as a specification language for the SOFA component model (Sect. 2.1), behavior protocols are also used in the Fractal component model (Sec. 2.2, project with France Telecom [12]).

Behavior protocols describe the behavior with respect to method calls on component boundaries (interfaces). Method's parameters, return values, and internal computations are abstracted.

2.3.1 Events and Protocols

We distinguish two types of an event: *request* and *response*. Together, the request and response events correspond to the notion of method invocation. As usual in process algebra, events are atomic, so that only one can be executed at a time.

An event is accosiated with a method name. With respect to software components, a method name consists of an interface name and a method identifier. Then, we denote an event (*event name*) by a method name following the event type mark: "↑" or "^" for request and "↓" or "$" for response. For example, an event name for a request of *open* method on $file$ interface takes the form $open.file\uparrow$.

From the component's point of view, an event can be *emitted*, *accepted*, or is *internal*, syntacticaly forming an *event token*. Event can be emitted either as a request on a required interface, or as a response on a provided interface. Event token denoting an emitted event is prefixed by "!" while an event token denoting acceptance of an event is prefixed by "?". Event which occurs between component's internal interfaces forms and event token prefixed with "τ". For example, emitting a response to the *commit* method on the *customer* interface takes the form $!customer.commit\downarrow$. A sequence of event tokens forms a trace. A set of all possible (allowed) traces forms a *behavior* (a language).

A *protocol* is a regular-like expression used for expressing a behavior. In addition to event tokens, it is composed of operators *sequencing* ";", *alterna-*

tive "+", *repetition* "$*$", *and-parallel* "$|$", and *or-parallel* "$\|$".

The result of the sequencing operator $A;B$ consists of the traces generated by a concatenation of all the traces generated by A by all the traces generated by B, while the result of the alternative operator $A+B$ consists of all the traces generated either by A or by B. For example:

$$\begin{aligned} L(a+b) &= \{< a >, < b >\} \\ L(a;b+c;d) &= \{< a,b >, < c,d >\} \\ L((a+b);(c+d)) &= \{< a,c >, < a,d >, < b,c >, < b,d >\} \end{aligned}$$

The repetition operator $A*$ generates all the traces produced by an arbitrary but finite repetition of all the traced generated by A. Note that zero-times repetition is allowed, resulting in an empty behavior *null*. For example:

$$\begin{aligned} L(a*) &= \{< null >, < a >, < a,a >, < a,a,a >, \ldots\} \\ L((a+b)*) &= \{< null >, < a >, < b >, < a,a >, < a,b >, \ldots\} \end{aligned}$$

And-parallel operator $A|B$ represents all the interleavings of event tokens in A and B. It is useful to express parallel execution of two traces. The or-parallel operator $A\|B$ is an acronym for $A+B+(A|B)$. For example:

$$\begin{aligned} L(a|b) &= \{< a,b >, < b,a >\} \\ L(a;b|c) &= \{< a,b,c >, < a,c,b >, < c,a,b >\} \end{aligned}$$

Several acronyms are defined for the most often used constructs (see Fig. 2.4 for examples): A simple method call $!i.m$ stands for $!i.m\uparrow;?i.m\downarrow$, a simple method acceptance $?i.m$ stands for $?i.m\uparrow;!i.m\downarrow$, a composite method call $!i.m\{A\}$ stands for $!i.m\uparrow;(A);?i.m\downarrow$, and finally a composite method acceptance $?i.m\{A\}$ stands for $?i.m\uparrow;(A);!i.m\downarrow$.

With respect to components, we distinguish the following specific behavior protocols:

- a *frame protocol* specifies the behavior of a particular component (Fig. 2.4), Typically, a frame protocol describes the expected behavior of a primitive as well as composite component and is written by a developer.
- an *architecture protocol* specifies the behavior of several components, usually constructed automatically from frame protocols of particular components, and
- an *interface protocol* specifies the behavior on a particular interface.

For composite component, both frame and architecture protocols exists.

For additional operators (*restriction* "/" and *adjustment* "$|$ $|$") and acronyms we refer the reader to the original paper [10].

2.3.2 Example

An example of a real-life frame protocol is presented in Fig. 2.4. The protocol displays the behavior of the `Arbitrator` component from the Airport Internet providing service example [15] (Fig. 2.5) developed within the scope of the Component Reliability Extensions for Fractal Component Model project [12]. The component starts with a synchronization with other components. First, it awaits a Start call from the `IArbitratorLifetimeController`. Then (;), it starts the initialization of the `ITokenLifetimeController` component by calling the Start method. The synchronization ends by accepting a response of the `Start` call on `ITokenLifetimeController` and emitting a response to the `Start` call on `IArbitratorLifetimeController`. Both calls are executed synchronously, which is denoted by square brackets. Synchronous execution (*atomic actions* [19]) is not a basic part of Behavior protocols, it has been introduced in the project to simplify complex synchronization of components at the beginning.

After the synchronization, the component runs in parallel (|) five protocols. The first one reacts on the `ILogin` interface (`?ILogin.`). It accepts the calls (+) of `GetTokenIdFromIpAddress`, `LoginWithFlyTicketId`, `LoginWithFrequentFlyerId`, `LoginWithAccountId`, and `Logout` methods. The implementation of the `GetTokenIdFromIpAddress` method does not produce observable events, thus the specification of the method is empty (there are no curly brackets). In contrast to the `GetTokenIdFromIpAddress` method, the `LoginWithFlyTicketId` method does produce observable events: First, it calls the `CreateToken` method on the `IFlyTicketAuth` interface. Then, the `DisablePortBlock` method is called on the `IFirewall` or (alternatively +) the method call is skipped ($NULL$). Similarly behaves the `LoginWithFrequentFlyerId`, `LoginWithAccountId`, and `Logout` methods.

Next three parallel protocols differs only in the indexes. The methods `TokenInvalidated_*` on the interface `ITokenCallback` calls the appropriate method `EnablePortBlock_*` on the `IFirewall` interface.

The last parallel protocol awaits for the `IpAddressInvalidated_1` method call on the `IDhcpCallback` interface. In the implementation of the method, the `InvalidateAndSave_2` method is called on the `IToken` interface.

All five parallel protocols are executed continuously since all can be repeated (∗).

2.3.3 Composition and Compliance

The *composition operator* ⊔ expresses the behavior of two communicating components. Formally, $A \sqcup_E B$ collects together two behavior protocols A and

```
( ?IArbitratorLifetimeController.Start^ ;
  !ITokenLifetimeController.Start^ ;
  [?ITokenLifetimeController.Start$,
    !IArbitratorLifetimeController.Start$] );
(
  (
    ?ILogin.GetTokenIdFromIpAddress +
    ?ILogin.LoginWithFlyTicketId {
      !IFlyTicketAuth.CreateToken ;
      (!IFirewall.DisablePortBlock + NULL) }
    +
    ?ILogin.LoginWithFrequentFlyerId {
      !IFreqFlyerAuth.CreateToken ;
      (!IFirewall.DisablePortBlock + NULL) }
    +
    ?ILogin.LoginWithAccountId {
      !IAccountAuth.CreateToken ;
      (!IFirewall.DisablePortBlock + NULL) }
    +
    ?ILogin.Logout {
      !IToken.InvalidateAndSave_1 }
  )* |
  ?ITokenCallback.TokenInvalidated_1 {
    !IFirewall.EnablePortBlock_1 }*
  |
  ?ITokenCallback.TokenInvalidated_2 {
    !IFirewall.EnablePortBlock_2 }*
  |
  ?ITokenCallback.TokenInvalidated_3 {
    !IFirewall.EnablePortBlock_3 }*
  |
  ?IDhcpCallback.IpAddressInvalidated_1 {
    !IToken.InvalidateAndSave_2 }*
)
```

Figure 2.4: Behavior specification of the Arbitrator component (Fig. 2.5) [15]

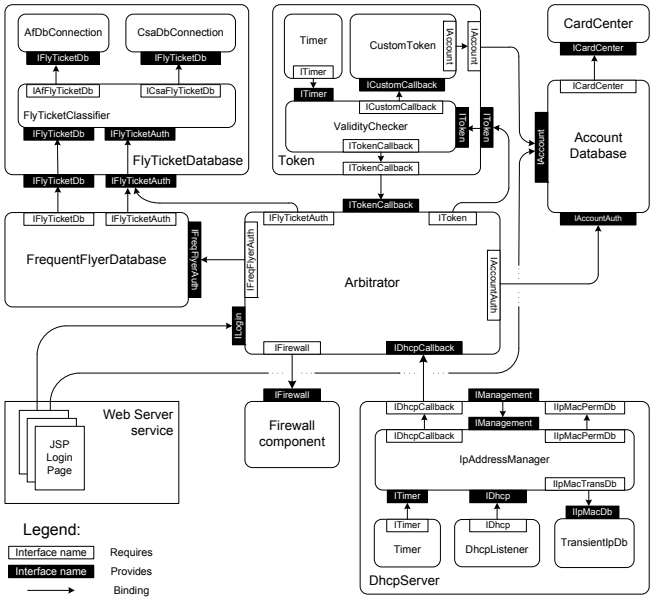


Figure 2.5: Architecture of the Airport Internet providing service [15]

B by arbitrary interleaving their events. Additionally to the similar parallel operator, the composition operator merges all the events of the form $!m \in A$ and $?m \in B$ (and vice versa) where $m \in E$ to the internal event τm.

Unfortunately, the composition operator does not express composition errors. Thus, the *consent* $\bigtriangledown$ operator has been introduced [13] (*horizontal contract*) by enhancing traces with *error tokens*. Three types of errors are detected:

- a *bad activity* occurs when an event is emitted, but not absorbed. The trace ends with the $!\epsilon$ token,
- a *no activity* (deadlock) occurs when no component can continue and at least one component has not finished. The trace ends with the $\oslash\epsilon$ token, and
- an *infinite activity* (divergence) occurs when components leads to infinite run, i.e. the component communication never stops.

Behavior compliance (or *compliance* for short) express the relation between the expected behavior specified by a developer and a real behavior specified by the automatically generated architecture protocol (a *vertical contract*). Intuitively, the real behavior should be a "subset" of expected behavior when require and a "superset" when provide. In a sense of behavior protocols, every absorbed method calls in the frame protocol should be absorbed in

the architecture protocol, while every emitted method call in the architecture protocol should be emitted in the frame protocol.

The exact semantics of compliance has evolved from naive [9] and pragmatic [10] to consensual [14], where the compliance is defined by a consent operator between an architecture protocol and an inverted frame protocol.

In the SOFA component model, the compliance is automatically verified by an automated tool, a protocol checker [16].

2.3.4 Extended Behavior Protocols

Recently, Kofron proposes in his PhD thesis [18] several extensions to Behavior protocols to simplify several constructs. In particular:

- *Local variables* and *method parameters*, but with enumerated types only. Local variables are introduced in order to eliminate a common practice of specifying a superset of behavior, which could be refined by additional data.

- *Multiple synchronization*, which are represented with specialized emit or accept actions which are shared among a set of protocols. The semantic differs from atomic actions as mentioned in Sect. 2.3.2 due to the problem of broken associativity of the composition.

- *Cycles* as a more powerful repetition construct than the standard repetition operator $*$. A new construct *while* is introduced which allows to repeat the protocol till the specified variable contains a specified value.

- *Switch statement* as a construct of conditional execution not presented in Behavior protocols at all.

Specifications in Extended Behavior Protocols (EBP) are supposed to be converted to the Promela [49] language and checked with the SPIN model checker [23]. Unfortunately, exceptions have not been proposed; probably due to difficult conversion to the Promela language.

Chapter 3

Related Work

3.1 Component Models

In this chapter, we introduce a selection of related component models along SOFA and Fractal presented in Chapter 2. Component-based system development promises significant advantages over traditional structural of object-oriented programming, including increased speed of development, reusability, and verifiability. Components are studied heavily in recent years, in such topics as versioning [78], benchmarking [79, 80], verification [20], deployment [81], performance prediction, and quality-of-service [86, 82]. For a recent comparison and evaluation of practical applicability of different component models, we refer the reader to the CoCoME contest project [87].

In the rest of the Chapter, we present the Tracta/Darwin component model (Sect. 3.1.1) which inspired many modern component models, and the Palladio component model (Sect. 3.1.2) as a representative of performance-prediction approaches.

3.1.1 Tracta/Darwin

Tracta [22] is a development framework whith associates behavior specifications with software components to analyze a system architecture. Architecture of the system is specified using Darwin [21]. Similarly to SOFA, Darwin distinguish *primitive* and *composite* components. Component services are published via *portals*. Portals are similar to interfaces, but provided and required are not distinguished. Darwin does not have connectors. Instead, connectors are a special class of components.

In contrast with SOFA, developers specify a behavior for primitive components only. The behavior is expressed by a finite state Label Transition System in a graphical notation. Alternatively, the behavior can be specified

textually via process notation of FSP (Finite State Processes). Thus, from the behavioral view, components are finite state processes. Connectors are modelled similarly to components.

Composite components are constructed by subcomponents. A behavior of composite components is then computed by a parallel composition of particular subcomponents.

Tracta allows verifying safety and liveness properties. Safety property is specified via property automata. These describe a set of allowed traces. If a model produces traces which are not included (accepted) in property automata, a violation is reported. Liveness properties are specified via Linear Temporal Logic.

3.1.2 Palladio

The Palladio Component Model (PCM) [83, 84, 85, 82] aims at a system quality-of-service (QoS) predictability. Its purpose is to estimate properties such as response time, throughput, and resource utilization early at design time. Following the idea of componetns presented in [67], PCM defines components as a black-box entity with well-defined interfaces.

The performance of a component varies among different machines it is deployed on. Obviously, while the designer of a component is aware of the component's requirements, he/she cannot predict its performance on an unknown hardware. PCM addresses the difficulty of performance prediction on varying environment by *parameterizing* the QoS characteristics over environmental influences.

The final system characteristics depends on the deploying environment and usage profile. These information are not typically available at the stage of component development and/or architecture design. Therefore, PCM distinguishes, along with *component developer* (implements and specifies the properties of the component) and *software architect* (leads the process of development, creates the *assemble model*), other *developer roles* involved in the system development: *deployer* (specifies hardware resources), *domain expert* (requirement analysis), and *QoS analyst* (extracts QoS requirements form the specifications, performs analysis, and estimates missing parameters). For each of these roles, a domain specific modelling language has been created and relations among them have been defined.

3.2 Process Algebras

Process algebras formally specify a behavior of processes and allow reasoning about concurrent systems. Behavior is a description of observable or internal actions a process perform, including their order and potentially other aspects as probability and time. Behavior protocols described in Sect. 2.3 belongs to process algebras.

A process can be modeled as an automaton, where transitions represent actions. A behavior is then a path from an initial state to some of final states. To express complex systems with concurrent and communicating processes, most of the algebras define a parallel composition.

Among most popular process algebras are CCS, CSP, ACP, and μ-calculus. We introduce CCS in detail.

3.2.1 Calculus of Communicating Systems

Mainly Robin Milner developed calculus of Communicating Systems (CCS). Most of CCS have been introduced in 1978 [54], including composition, message passing, and internal actions prefixed by "τ". Lately, observational equivalence and strong equivalence was formulated [55] and CCS as a complete algebra has been introduced [56]

We present concepts of CCS as stated in [68]. A basic behavior entity in CCS in an *agent* (also a *process*). An agent performs *actions* (or *events*) which is either a communication with another agent or it occur independently. An action can be observable (reception or transmission denoted by a line $\overline{a}$) or silent (internal) τ.

For example, an agent *Repeater* may be defined on a set of actions $\{in, out\}$. A sequence of events is denoted by an *action* operator ".". If a is an action and A is an agent then $a.A$ is an agent. The action operator is also referred as the *prefix* operator with the meaning "an agent A is active only after the action a has been performed" and denoted as $a.A \xrightarrow{a} A$.

Thus the behavior of the *Repeater* agent may take the form of

$$Repeater \stackrel{def}{=} in.\overline{out}.Repeater$$

One of the simplest agent is 0 (read as *nil*) which is incapable of any actions.

The exact names of the actions are not significant; they only suggest the underlining meaning in the real word. CCS allows to rename actions of the agent by the *relabelling* operator $R[f]$, where R is the agent name and f is the relabelling function. For example, relabelling the actions in the *Repeater*

agent may take the form of

$$Repeater^2 \stackrel{def}{=} Repeater[incoming/in, outcoming/out]$$

defining a new agent $Repeater^2$ that is similar to the *Repeater*, but uses the action *incoming* instead of *in* and *outcoming* instead of *out*. Relabelling is used in agent replication, where exact action names would lead to unintentional communication confusion among replicas.

An agent may choose of several alternative actions via the *choice* + operator. If A and B are agents then $A + B$ is an agent. Choosing to perform A preempt the performing of B and vice versa.

Complex systems consist of many independent agents, which interact with each other. In CSS, a combination of two agents is constructed via the *composition* operator $|$. If A and B are agents then $A|B$ is an agent. Two agents may interact by a convention via same action names — *ports*. For example, consider the agent C defined as a composition of agents A and B:

$$\begin{aligned} A &\stackrel{def}{=} a.x.A \\ B &\stackrel{def}{=} \overline{x}.b.B \\ C &\stackrel{def}{=} A|B \end{aligned}$$

Agents A and B are willing to communicate over the shared action x. However, the composition does not specify (limit) that these two components must interact with each other; the resulting agent C can communicate with other agents via x and $\overline{x}$ actions as well.

If we want to limit the scope of actions, we must use the *restriction* operator $\backslash\{s\}$. Then, agents are synchronized via the specified actions s, which becomes an internal action τ. For example, the behavior of the agent defined as:

$$D \stackrel{def}{=} (A|B) \setminus x$$

has no actions x or $\overline{x}$.

Formal definitions of basic operators are usualy expressed using Structural operational semantics developed by Gordon D. Plotkin via inference rules. An Inference rule consists of a set of promises, an optional condition, and a conclusion:

$$\frac{premise_1 \; premise_2 \; \ldots \; premise_n}{conclusion} \; condition$$

An action operator is the only axiom:

$$\frac{}{a.A \stackrel{a}{\rightarrow} A}$$

The composition operator (two symmetric rules):

$$\frac{A \xrightarrow{a} A'}{A|B \xrightarrow{a} A'|B}$$

$$\frac{B \xrightarrow{a} B'}{A|B \xrightarrow{a} A|B'}$$

The choice operator (two symmetric rules):

$$\frac{A \xrightarrow{a} A'}{A + B \xrightarrow{a} A'}$$

$$\frac{B \xrightarrow{a} B'}{A + B \xrightarrow{a} B'}$$

The communication rule (the communication happens only when two compatible actions are performed) takes the form of:

$$\frac{A \xrightarrow{a} A' \; B \xrightarrow{\overline{a}} B'}{A|B \xrightarrow{\tau} A'|B'}$$

The restriction rule:

$$\frac{A \xrightarrow{a} A'}{A/L \xrightarrow{a} A'/L} \; a, \overline{a} \notin L$$

The relabelling rule:

$$\frac{A \xrightarrow{a} A'}{A[f] \xrightarrow{f(a)} A'[f]}$$

To capture differences between agents, equivalence relations have been introduced. Milner discuss several approaches to agent equivalence, including the similarity of behavior and derivation tree. Intuitively, two agents should be equal if an external agent communicating with them cannot recognize the distinction between them. We present two concepts of equivalence: a *trace equivalence* and a *bisimulation equivalence*.

If the finite traces of the agents A and B are identical, the agents A and B are trace equivalent, denoted as $A \approx_{tr} B$. Trace equivalence is suitable for verifying safety properties of systems specified by visible actions.

The agent A is strongly bisimilar to the agent B if and only if there exists a bisimulation relation R over agents such that for all actions a the following holds:

(i) $(A, B) \in R$,

(ii) if $(A', B') \in R$ and $A' \xrightarrow{a} A''$ then there is B'' such that $B' \xrightarrow{a} B''$ and $(A'', B'') \in R$, and

(iii) if $(A', B') \in R$ and $B' \xrightarrow{a} B''$ then there is A'' such that $A' \xrightarrow{a} A''$ and $(P'', Q'') \in R$.

We can always replace an agent with a strongly bisimilar one.

3.2.2 π-calculus

The π-calculus [69] is a continuation of CCS supporting a dynamic configuration change during the computation. The reconfiguration is accomplished by passing a port's reference (*mobility*), replication (spawing/forking) of processes, and a creation of a new name (port).

3.2.3 Communication Sequential Processes

The theory of Communication Sequential Processes (SCP) has been developed by Tony Hoare. First introduced in 1980 [59] based on synchronous communication and as a guarded command language [58], later based on trace theory [57] and polished [60].

3.2.4 Algebra of Communicating Processes

Algebra of Communicating Processes (ACP) was introduced in 1980 [61] strictly as an algebra with alternative, sequential, and parallel composition and no communication, later including communicaton [62] as well.

3.3 Program Slicing

A *program slice* is a restriction of a program that is essential to values according to *slicing criterion*. A slicing criterion consists typically of a program location and a set of variables. *Program slicing* is then a process of computation of a program slice according to the specified slicing criterion.

Program slicing was introduced in 1979 by Weiser in his PhD thesis [35]. Initially, program slicing should correspond to the programmer's point of view when debugging a program [37, 36]: if there is an incorrect value in a variable at some point (positon), the bug is probably in the (smaller) program slice with respect to the variable and the position. Thus a program slice should be constructed by removing statements from the original program, should be syntactically correct, and executable. However, since there are many different

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv)
4  {
5    FILE *f = fopen( argv[ 1], "r");
6    int c0 = 0;
7    int c1 = 0;
8    int i;
9    while (fscanf( f, "%d", &i) == 1)
10   {
11     if (i<0)
12       c0++;
13     if (i>0)
14       c1++;
15   }
16   printf( "%d\n", c0);
17   printf( "%d\n", c1);
18   fclose( f);
19
20   return 0;
21 }
```

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv)
4  {
5    FILE *f = fopen( argv[ 1], "r");
6    int c0 = 0;
7
8    int i;
9    while (fscanf( f, "%d", &i) == 1)
10   {
11     if (i<0)
12       c0++;
13
14
15   }
16   printf( "%d\n", c0);
17
18   fclose( f);
19
20   return 0;
21 }
```

Figure 3.1: Original and statically sliced program according to c0

perspectives of view, a slicing criterion and a program slicing vary among applications. For example, some researches consider a program slice as a subset of the original program, not necessarily executable.

Nowadays, along debugging [65], program slices are used for program analysis, testing [72, 73], improving comprehension [74, 75], reducing costs of maintenance and evolution [66], and compiler tuning [38].

As an aside, an optimal program slicing (minimalization) is in general undecidable.

3.3.1 Static Slicing

Computing a program slice for any input with only static information is called *static slicing*. Figure 3.1 (left) shows an example of a program counting a number of positive and negative numbers in a file. A program slice according to line 16 and variable `c0` takes the form of Fig. 3.1 (right): all statements that do not influence a value of `c0` have been removed. In particular, computing of positive numbers in `c1` is unnecessary.

There are several approaches to how compute a program slice. In the original work of Weiser, a set of relevant statements is computed closely to transitive closure (result is a fixpoint), with respect to data and control flow. Alternatively, program slice can be reinterpret in terms of reachability in a Program Dependence Graph (PDG) [63]. PDG is a graph where vertices represent statements and predicates and dependences are represented by oriented

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv)
4  {
5    FILE *f = fopen( argv[ 1], "r");
6    int c0 = 0;
7    int c1 = 0;
8    int i;
9    while (fscanf( f, "%d", &i) == 1)
10   {
11     if (i<0)
12       c0++;
13     if (i>0)
14       c1++;
15   }
16   printf( "%d\n", c0);
17   printf( "%d\n", c1);
18   fclose( f);
19
20   return 0;
21 }
```

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv)
4  {
5    FILE *f = fopen( argv[ 1], "r");
6    int c0 = 0;
7    int c1 = 0;
8    int i;
9    while (fscanf( f, "%d", &i) == 1)
10   {
11
12
13
14       c1++;
15   }
16   printf( "%d\n", c0);
17   printf( "%d\n", c1);
18   fclose( f);
19
20   return 0;
21 }
```

Figure 3.2: Original and dynamically sliced program according to the line 17, c0 and c1, and input $< 2, 5, 6 >$

edges. Slicing criterion is then a selected vertex v and a slice is a subgraph reachable from v.

3.3.2 Dynamic Slicing

Moreover, we can further slice a program by taking a fixed input into account — *dynamic slicing*.

Then, a slicing criterion is enhanced with *input* and a program location is replaced by a statement occurrence. An example of dynamic slice is depicted in Fig. 3.2 with respect to the first occurrence of (statement on) the line 17, variables `c0` and `c1`, and input $< 2, 5, 6 >$. Because all the input numbers are positive, the condition on line 11 is always false and thus redundant, while the condition on line 13 is always true. Note that static slice can remove neither one of the conditions, since the real numbers from the input are unknown.

For a comparison of program slicing techniques and history remarks, please refer to the Tip's survey [64].

3.4 Parallel and Distributed Model Checking

3.4.1 Introduction

In a last decade, many researches focused on parallelizing and distributing model checkers to cope with large state spaces. There are several reasons why: The increasing frequency of processor raises problems with cooling and power consumption, therefore the development of processors continues with increasing the number of computational units (processors) rather than increasing a frequency. Contemporary computers are embedded with at least two computational units and thus a single sequential computation performed on it "wastes" about a half of the available performance.

The increasing demand for computer interconnection (particularly caused by the popularization of the Internet), causes a rapid progress in technology of local networks. For example, the most popular local area network Ethernet (IEEE 802.3 standard) transmits data at the speed 10 Mbps in the year 1980, 100 Mbps in the year 1995, 1 Gbps in the year 1999, and 10 Gbps is in the standardization process (IEEE 802.3ae). Gigabit Ethernet is usually embedded in contemporary computers by a manufacturer.

A large state space causes two main problems. The first one is the time required to traverse (generate) the whole state space. We would like to reduce runtime requirements by enhancing the computational power by aggregating available CPUs.

The second problem is a storage area for generated states. Traversing algorithms requires saving visited states to avoid re-entering explored parts of the state space. This sets a limitation to a state space size. By aggregation available memory, we would like to push further this limitation.

Because there is an inconsistency in the terms parallel and distributed, we have to explicitly emphathetize the difference. By *parallel*, we refer to multi-threading algorithms, typically in the environment of shared memory and symmetric multi-processing. On the other hand, by *distributed*, we refer to a computation on different computational nodes connected via network interfaces. Such an environment is characterized by a high access latency to other's memory in contrast to shared memory.

3.4.2 Requirements

Ideally, the distributed model checking algorithms should:

- Effectively aggregate the available computational power. Thus the total computational power should be proportional to the sum of power of all

the participated computational nodes in the network (cluster). Particularly, all the processors should be fully loaded during the verification.

- Effectively aggregate available memory. Similarly to the computational power, the total amount of states that can be stored in the memory should be proportional to the sum of memory of all the computational nodes in the cluster.
- Require only low-cost synchronization in parallel computing.
- Require low network bandwidth, i.e. by minimizing inter-node communication.
- Allow to effectively utilize external storage devices.

These characteristics should be preserved even in non-homogeneous clusters, where the actual performance parameters and available resources differ among computational nodes. Furthermore, the algorithm should cope with variing available power usual in a network of user workstations.

3.4.3 Hash-based Algorithm

One of the first approaches to distribute verification was introduced by Stern and Dill [31] for the Murφ verifier [33]. Because the hash-based algorithm becomes widely adopted and is used almost exclusively, we describe the work in detail and later discuss its potential weaknesses and a scope for improvements.

States are stored in a hash table (the state table) which is partitioned over the computational nodes in the cluster (a node becomes an owner of a particular part of the state space). Each node keeps a queue of unexplored states. A newly generated state is passed to the hash function. The result of the hash function points to the owner of the state. If the state belongs to the actual node, then it is processed locally, otherwise the message passing layer sends the state as a message to its owner. Received states are tested whether they have been reached before according to the state table. New states are added to the queue of unexplored states.

The search is finished when there are no messages in process and working queues of all the nodes are empty. The master node, which queries all the nodes for the number of states in the queue and the number of messages sent and received, tests this condition.

The error trace is generated by the node, which reaches an error state. During the verification, each node writes information about the predecessor of

the newly processed state to a local file. The error trace is then reconstructed by a backward search through these (now distributed) files. The authors also employ a message aggregation technique to improve network utilization.

The distribution of the work is achieved by the randomness property of the hash function. A good hash function guarantees a relatively good state distribution.

3.4.4 Identifying and Addressing the Problems

Although the most used, the hash-based approach suffers from several problems with respect to ideas presented in Sect. 3.4.2.

States and Transitions

Surprisingly at first sight, the real workload is defined by transitions and not states. Because the method ignores the density of the graph, some non-homogeneous models may lead to load asymmetry. For example, a node owning an exceptional node with a high-degree of incoming transitions will be overloaded by incoming messages. However, because such pathological instances are rare, the problem is not addressed by traditional distributing methods and states are considered as a good approximation of transitions.

Network Bandwidth and Load-balancing

In the original paper [31], the authors point out a problem of high bandwidth demand. A partial solution proposed is sending a (small) result of a hash function on a state instead of the whole (large) state and wait for one-bit response. In such a case, the owner of the state cannot expand it.

Another solution to the problem was presented by Lerda and Sisto [30] when distributing the SPIN [23] model checker. As discussed, optimal partitioning function should satisfy three conditions. First, a region selection should be computed using information obtained from the state only. Second, states should be spread across regions evenly to guarantee balanced workload. Finally, a number of transitions crossing region boundaries should be minimized to minimize a required network bandwidth, i.e. the partitioning function should exploit a structure of the state space.

Exploiting the structure of a state space is extremely difficult, in general as hard as the model checking itself. Especially in real-life industry specifications, the generated state space is very non-uniform, with both dense and sparse regions.

Lerda and Sisto's suggested partitioning function to select regions according to a specified process of the specification. In the Promela language [49], a system is specified via synchronously communication processes. In one step (a single transition), the system differs from one state to another only in few components. Typically, this is either a simple internal action where states of another components are untouched, or it is a communication which affects states of two components only. Thus, a transition crosses a region boundary only when communicating with the specified process, all other transitions are processed localy. Unfortunately, the method is very sensitive to the actual structure of the specification.

Another approach by Ciardo et al. partitions a state space statically according to user-defined hash functions [25]. Although a well-defined hash function partitions the state space evenly, the main drawback of the method is that the user has to fully understand the state space structure and then specify a function which both spreads the states evenly and simultaneously minimizes the number of cross-transitions. This is obviously not an easy task. Moreover, when a single computational node becomes overloaded and starts to trash, the whole computation stalls.

This approach was later improved by Nicol and Ciaro [26] with heuristics. Before the verification itself, the state space is examined by a random walk. Given explored states, a search tree is constructed using a randomized lexicographical function and is duplicated across nodes. Exits from the search tree define classes of states, which are associated with nodes.

The method is to some extent adaptive to the actual structure of the state space, however the structure is static during the verification and is not able to adapt to actual CPU or memory load. This is addressed by dynamic remapping, where the current load (number of states) is periodically checked and classes of states are tuned accordingly. The dynamic load balancing is considered essential.

Heyman et al. [50] present a method of dynamic memory-balancing by an adaptive partition function. When memory requirement becomes unbalanced, a balance procedure is executed. The coordinator (a single process in the network) matches processes with high memory requirements to those with small one. Then, the partition function is modified, the state space is re-partitioned, and finally all other processes are informed about the new partitioning function. The main drawback of this approach is that during the re-partitioning, the affected processes cannot continue with the state space traversal.

A dynamic load-balancing is a nontrivial task. We illustrate the problem on the work of Behrmann et al [52]. While the distribution performed

well on a platform providing fast communication [51], significantly worse performance was achieved on a Beowulf cluster [52] caused by load-balancing problems and high communication overhead. The author identified the problem in overreacting of the auto-balancing system to small load differences, leading to unstable system where the load of one or more nodes drops to zero. An interesting observation is that an increase in the size of the output collecting buffers causes worse performance. The reasons are that the increasing latency between nodes delays load-balancing algorithm, makes worse the approximation of the breadth-first search order, and reduces the coalescing effect of explored states leading to more explored states.

Note that the problem with network bandwidth is proportional to the speed of a new state generation, the size of a state, and the efficiency of state encoding.

Non-homogeneous Clusters

Intuitively, effectively distribute a work in clusters of computers with different hardware using hash function is even more complicated than in uniform clusters. The difficult task is to decide which parameter should be chosen to distribute the work. If it is the CPU performance, fast CPUs are prone to exhaust available memory and start to trash. On the other hand, partitioning by available memory causes situations where fast computers are not fully loaded.

Non-homogeneous environment is usually not addressed. Most common approach is to divide the state space according to available memory which is main limitation when coping with large state spaces.

Non-dedicated Clusters

Small as well as large organizations own tens or hundreds of workstations used by administration. These computers are idle most of the time and available for distributed computing. However, the algorithms must cope with concurrently running user processes and thus varying available performance. This situation is somehow similar to non-homogeneous clusters and is rarely addressed by researches.

Jones and Sorber [34] present the bee-based error exploration (BEE) algorithm for finding LTL [76] violations, suitable for clusters of workstations. The algorithm works by coordinating parallel random walks. In each step, the algorithm decides under certain probability whether to backtrack or continue by exploring one of successor states. Backtracking is implemented as a random choice of one of the states from the search stack. Unfortunately, the

BEE algorithm does not assure exhaustive exploration of the state space and thus is suitable for fast bug-hunting rather than exhaustive model-checking.

3.4.5 External Model Checking

Large storage capacity is attractive for model checking since it allow to explore significantly larger state space than is allowed when using internal memory only. Unfortunately, when comparing with internal memory, mass storage devices suffer from high access latency. For example, the access time of modern hard-disks is usually more than 6ms [53]; that is from 10^5 to 10^6 times slower than the internal memory. Thus, employing external devices requires specific data organization with respect to locality — randomization of states by partitioning function almost prevents employing external memory effectively in a hash-based distributed model verification. While for most sequential model-checkers DFS ordering is easier to implement than BFS, in distributed environment, DFS implementation is complicated by its sequential nature [27].

Among the first approaches for external model checking (non-distributed) is the work of Stern and Dill [32] in Murφ [33]. The state space is searched in the order of BFS. In contrast with DFS, newly generated states do not have to be filtered against the already explored states immediately. Instead, the filtering is processed after the whole level of states has been fully generated. Then, the table of explored states is retrieved form hard-disk *linearly*, thus mitigating latency time.

Surprisingly, only little work has been done on distributed external model checking. As an example, we present details about recent work of Jabbar and Edelkamp [24] on distributing the External A* [28] algorithm in IO-HSF-SPIN [29].

The external A* algorithm stores the search horizon on external device (BFS ordering). States are organized in buckets with the same path length (g) and heuristic estimate (h). Since heuristic estimate groups similar states into same bucket, duplicate detection can be limited to particular buckets.

The parallelization is based on an observation that the internal work on each state bucket can be parallelized on more processors. On each of the processors, several concurrent processes are deployed. A work queue is maintained via shared file, constructing a communication channel. A job consists of a bucket index and a file to be considered.

Each bucket of states proceeds over four phases. In the *exploration* phase, successors are saved in a file (g, h, p) where g and h are hash values and p is a processor index. In the *first sorting* phase, each processor sorts its file. In the *distribution* phase, one processor distributes states in all files according to a hash value. Until the distribution finishes, all other processors have to

wait. Note that this is a potential bottleneck. Finally, in the *second sorting* phase, processors once again sort their file to further eliminate duplicates.

Experiments on a multiprocessor machine with 3 processors reveal a speed-up factor from 1.64 to 2.41 depending on the task. On two machines connected via nfs, the speed-up was from minimal 1.08 to 1.41. The bottleneck was a saturated nfs connection.

Chapter 4

Commented Overview of the Papers

In this chapter, we present a basic overview of selected papers included in Chapters 5-10.

Exceptions in Component Interaction Protocols - Necessity (Chapter 5)

In the paper, we present an analysis of the role and importance of exceptions in a behavior specification and the way we have enhanced Behavior protocols by them to handle exceptions in an efficient way in terms of readability, comprehension, and the size of the specification.

Most of the current component protocols do not allow to express exceptions explicitly. To illustrate viability of our results, we present a study of applying different techniques of expressing exceptions in Behavior protocols in a specification of a real-life component-based application which uses exceptions heavily. Without explicit exceptions, the specification becomes overly large, complicated, hard to understand, and error-prone.

The paper addresses the goal (1) stated in Sect. 1.2. The author of the thesis identifies the problem, and proposes the solution and the syntax. The co-author – Frantisek Plasil – fine-tuned the behavior of parallel exceptions and polished the syntax.

Reducing Component Systems' Behavior Specification (Chapter 6)

In this paper, we present a method of specification reduction based on symbolic manipulations. The method, which is to be applied at the input of a verification tool, consists of an iterative application of reduction rules. The reduction rules modify the specification while preserving important characteristics of the model with respect to the composition (the consent operator and the inverted frame protocol). If the result of the reductions identifies a composition error, a corresponding counter example is back interpreted in the original specification. This makes the reduction method fully transparent to the user.

To guarantee correctness of the reductions, we have formulated several conditions to be satisfied. Reductions are of a low overhead, since the algorithm is linear with the size of the specification.

Although a "full" reduction of the specification is not guaranteed in general, real-life case studies, such as the one presented in the paper, show that most of such specifications contain several typical patterns to which reduction rules apply (and allow a full reduction).

Due to page number limitations, the example presented in the paper is truncated. We elaborate the example fully in detail in Appendix A.

The paper addresses the goal (1) stated in Sect. 1.2. The author of the thesis investigates the possibilities of symbolic manipulations, designs the reduction rules and elaborates the proof-of-the-concept examples. The co-author – Frantisek Plasil – reviewed the reduction rules and proposes the structure of reduction explanations.

On Distributed Verification of Generalized Interaction Models of Software Components (Chapter 7)

In the paper, we sketch a concept of distributed model verification based on interfering automata, a generalized language suitable for describing the behavior of complex interacting systems based on Labelled Transition Systems. The work is included as a connection between the distributed model verification presented in Sect. 8 and interfering automata presented in Sect. 9.

The paper addresses the goal (2) stated in Sect. 1.2.

Streaming State Space: A Method of Distributed Model Verification (Chapter 8)

In the paper, we introduce a novel method of distributed model verification of safety properties for large state spaces, targetted at common network infrastructures of non-dedicated workstations, proposed as an alternative to traditional partitioning-based approaches. Instead of dividing the state space into parts, we keep the states ordered and organized as a stream which is handed over in a logical circle across the computational nodes. Each computational node updates the stream in the part it sees.

Due to the predictable organization of states, parts of the state space can be temporarily stored on external high-latency storage devices.

The presented results show that the method scales very well with the number of processors and computational nodes involved. Additionally, the streaming method fits very non-uniform clusters and allows balancing the load dynamically during the verification.

The paper addresses the goal (2) stated in Sect. 1.2. The author of the thesis proposes the method, implements the proof-of-the-concept checker, obtains experimental results, and evaluates them. The co-author – Petr Tuma – suggests improvements in the argumentation and polished the text.

Identifying Representatives for Interfering Automata (Chapter 9)

The paper covers a technique of a partial-order reduction on generalized interaction protocols — Interfering automata. The technique substitutes a block of states by a single state — a *representative* — on-the-fly during the verification.

A representative is identified in each step of the exploring algorithm. Instead of a precise computation of covered states, which would be ineffective, we select only a subset of available transitions to continue in traversing. The method is suitable for distributed model verification, since statuses of the neighboring, potentially dislocated, states are not necessary.

The paper addresses the goal (3) stated in Sect. 1.2.

Implementation of a Linux Log-Structured File System with a Garbage Collector (Chapter 10)

The paper covers an implementation of a file system based on a data log organization (LFS).

An interesting characteristics of the log — with respect to the fault-tolerant verification — is its late (lazy) removal of deleted data and the concept of *snapshots*. Data to be deleted are not cleared immediately. Instead, they are kept until most of the free disk blocks are allocated. Then, the kernel of the operating system calls the *garbage collector* (a user-space process) to clean a part of the disk.

The file system can be employed in model checking in the following way. Traditionally, the model checker has to store the states in a file on a hard drive. When the states are stored sequentially, the hard drive is used efficiently. Unfortunately, removing of already explored states is very time consuming — states tend to be spread. This problem is solved almost exclusively in external model checking, where, at every moment, the model checker works with a set of files (the working set). In the recovery mode after a crash, the model checker has to reconstruct the working set. This is relatively easily achieved with a very important feature of the LFS — a *snapshot*. At a given time (typically after finishing a level in the BFS), the model checker calls the command of the operating system to create a snapshot. A snapshot consists of all the data at the specified time. Then, all the modification are stored in different areas of the disk so that the snapshot is not affected.

The streamed method presented in Chapter 8 does not use the traditional working set of files. Instead, the states are stored in linear input/output buffers. After the crash, the model checker has to recover the full single epoch. Because the LFS "does not forget" data, this is relatively easy. However the size of the stream increases during the verification, thus it would be inefficient to kept on a disk all the epochs. This is the place where the garbage collector helps. Since it is a user-space program, it is possible to "tweak" it in such a way that data of the old epochs are removed in preference.

The paper addresses the goal (2) stated in Sect. 1.2. The author of the thesis supervised the software project as a lecture in the Faculty of Mathematics and Physics, Charles University and polished the text.

Chapter 5

Exceptions in Component Interaction Protocols - Necessity

František Plášil and Viliam Holub

Contributed paper at **Architecting Systems with Trustworthy Components** seminar [4].

In *Architecting Systems with Trustworthy Components*,
published by Springer-Verlag,
Lecture Notes in Computer Science vol. 3938,
pages 227–244,
ISSN 0302-9743,
ISBN 3-540-35800-5,
Jan 2006.

Exceptions in Component Interaction Protocols - necessity*

Frantisek Plasil[1,2] and Viliam Holub[1]

[1] Charles University, Faculty of Mathematics and Physics,
Department of Software Engineering
Malostranske namesti 25, 118 00 Prague 1, Czech Republic
{plasil,holub}@nenya.ms.mff.cuni.cz,
WWW home page: http://nenya.ms.mff.cuni.cz/
WWW home page: http://nenya.ms.mff.cuni.cz/~holub/
WWW home page: http://nenya.ms.mff.cuni.cz/~plasil/
[2] Academy of Sciences of the Czech Republic
Institute of Computer Science
plasil@cs.cas.cz
WWW home page: http://www.cs.cas.cz/

Abstract. At ADL level, most of the current interaction protocols designed to specify components' behavior at their interfaces do not allow to capture exceptions explicitly. Based on our experience with real-life component based applications, handling exceptions as first class entities in a (formal) behavior specification is an absolute necessity. Otherwise, due to the need to capture exceptions indirectly, the specification becomes very complex, therefore hard to read and, consequently, error-prone. After analyzing potential approaches to introducing exceptions to LTS-based interaction specification (expressed via terms/expressions) in ADL, the paper presents the way we built exceptions into the behavior protocols. Finally, we discuss the positive experience with applying these exception-aware behavior protocols to a real-life Fractal component model application.

1 Introduction

There are many approaches to describe the desired behavior of software components. They include interface automata[4], behavior protocols[16], DFSM[19], usage policies[6], interactions and reactions[25], parametric contracts[18], UML2.0 State Machines (in principle stemming from Harel diagrams[8]) and Protocol State Machines[12], and CSP-based mechanisms, such as Wright[5] and FSP[11].

Those of them which are based on LTS (Label Transition System) where the transitions model atomic actions, allow for some kind of reasoning on behavior (e.g. equivalence[5], compatibility[4], compliance[16]). For instance, these atomic actions model the request and response triggered by a method call -

* This work was partially supported by the Grant Agency of the Czech Republic project 201/06/0770; the results will be used in the OSIRIS/ITEA project.

i.e. mostly the "control-observing" behavior of a component. Obviously, those LTS-based behavior description mechanisms which are directly applicable in architecture description languages cannot explicitly utilize any kind of diagrams, and therefore typically employ some kind of term expressions. However, there is a problem with this approach: capturing exceptions. While in a graphically expressed transition system, an exception can be expressed by adding another transitional edge, most of the term-expression based formalisms do not allow this easily. We encountered the problem when we were trying to employ behavior protocols [1,2,3,16], in non-trivial case studies of component behavior specification, comprising over 20 components each.

This paper aims at achieving two main goals:

1. To present a "reasonable" syntax extension of behavior protocols which does not violate the inherent regularity of the traces generated by the protocol (and therefore preserves important properties like protocol compliance decidability).
2. To show that the proposed syntax increases readability and significantly simplifies a behavior protocol when an exception is to be thrown/handled. This claim is supported by experimental results.

The paper is structured as follows: Sect. 2 describes the background - behavior protocols, in Sect. 3, the problem of handling exceptions in protocols is analyzed from a perspective of component communication and a solution is proposed. Section 4 illustrates the proposed solution on a case study. Section 5, as a part of overall evaluation, shares with the reader the experience with applying the proposed approach to a real-life Java project. Finally, Sect. 6 is focused on related work and Sect. 7 draws a conclusion.

2 Background - behavior protocols

The basic idea of behavior protocols can be illustrated on the following example (Fig. 1).

The picture shows the internal structure of a hypothetical Reservation component which is composed of five sub-components - Ticket manager (responsible for registration of tickets), Database manager (implementing the database behavior), Storage (permanently stores data), VISA (for payment authentication) and Operator verification (a connection to third-party servers).

Via behavior protocols, we can capture communication among these components. There are three types of protocols - *frame protocol* specifying the expected activities on components' boundaries (their frame), *architecture protocol* created automatically as a parallel composition of the frame protocols of the subcomponents (at the first-level of nesting) and the *interface protocol* describing the behavior only on a selected interface.

These abstractions allow for addressing two aspects of "design by contract": (i) *Horizontal contract* "Do the children cooperate with no conflicts?" (a conflict is statically detected as a *composition error*), and (ii) *vertical contract* "Do the

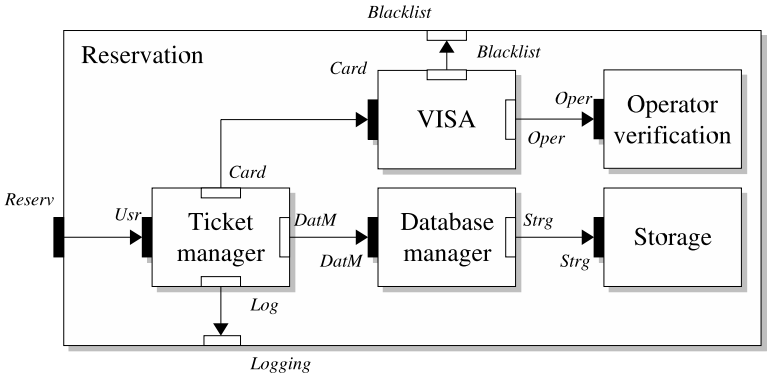


Fig. 1. The Reservation component

cooperating children do what the parent expects?" which is statically verified via evaluating *compliance* of the architecture protocol (determined by the subcomponents) and the frame protocol of the parent component. As an aside, the static composition error detection and compliance verification is done by a tool, protocol checker, available as a part of the SOFA project[21].

As to (i), three types of composition errors are identified - *bad activity* (an emitted event is not accepted), *no activity* (deadlock), and *divergence* (infinite activity). Definition of the semantics of compliance is crucial and has evolved from a *naive*[17] and *pragmatic*[16], to *consensual*[2] based on the idea that the architecture should work well (without composition errors) when cooperating with a separate component representing the architecture's environment. The behavior of this environment is defined as the "inverted" frame protocol of the parent component[2].

For example, compliance of the architecture protocol specifying the composed behavior of Ticket manager, Database manager, Storage, VISA and Operator verification with the frame protocol of Reservation can be verified. The frame protocol of the Ticket manager component can take the form:

```
?Usr.init;
(
  ?Usr.buyTicket {
    !DatM.preReserve;
    !Card.lookUp; !Card.payment;
    !DatM.reserve; !DatM.commit;
    !Log.print}
  +
  ?Usr.returnTicket {
    !Card.revert;
    !DatM.free; !DatM.commit}
)*;
?Usr.finish
```

The frame protocol specifies that the component expects (?) an init method call on the interface Usr followed (;) by alternatively (+) a call of buyTicket or returnTicket. After this is repeated a finite number of times (*), a finish call is accepted; no other incoming calls are allowed. The statements of the form ?i.a{P}, where P is a subprotocol, i is an interface name and a is a method name, is an abbreviation of ?i.a;(P)!i.a. The ?i.a means *accepting* (?) a *request* () i.a, and the !i.a means *emitting* (!) a *response* () to i.a. Along these lines we see that the buyTicket method (acquiring and reserving a ticket for the user), calls (!) the preReserve on the DatM interface to inform the Database manager that a ticket is to be reserved. As a next step, calls of lookUp and payment methods on the Card interface are made to perform the payment; further, the reserve and commit methods on the DatM interface are called in order to confirm the transaction. The last action is to print information about the transaction - Log.print.

Instead of the alternative operator +, we could use the *or-parallel operator* || to express that calls of buyTicket and returnTicket might be accepted simultaneously. Additional operators and further details are described in [16].

3 Handling exceptions in behavior protocols

3.1 Primitive techniques

In real settings, exceptional situations (not described in the previous protocol) also have to be handled - e.g. the VISA component may deny service due to a network error, and the Database manager may refuse to allocate appropriate resources. In other words, specifying behavior of a component inherently involves exceptions. However, expressing exception via the standard operators is tedious. For illustration, consider the DatM.preReserve method call from the example above which could throw a preReserveException exception. In order to specify this behavior, we have to split the return from the preReserve method into a regular return (?DatM.preReserve) and an accepting return with exception (?DatM.preReserveException) - we call this technique *intrinsic exceptions* handling. However, in consequence, the frame protocol length would expand rapidly (exponentially in the number of methods throwing an exception).

Below is a fragment of the frame protocol of Ticket Manager where several exceptions are thrown and handled - it illustrates how the protocol becomes complex. In the example, we suppose that the DatM.preReserve method can throw PreReserveException, methods Card.lookUp and Card.payment can throw NetworkException, and finally the DatM.reserve method can throw ReservationException.

Technique: Intrinsic exceptions

```
...
?Usr.buyTicket ;
  !DatM.preReserve ;
  ( ?DatM.preReserve ; !Card.lookUp ;
    ( ?Card.lookUp ; !Card.payment ;
      ( ?Card.payment ; !DatM.reserve ;
        ( ?DatM.reserve ; !DatM.commit ;
          ( ?DatM.commit ; !Usr.buyTicket )
          +
          ( // exceptions of DatM.commit
            ?DatM.DatabaseException;
            !DatM.cancel; !Card.revert; !Log.print;
            !Usr.buyTicket
          )
        )
        +
        ( // exceptions of DatM.reserve
          (?DatM.DatabaseException+?DatM.ReservationException);
          !DatM.cancel; !Card.revert; !Log.exEvent; !Log.print;
          !Usr.buyTicket
        )
      )
      +
      ( // exceptions of Card.payment
        ?Card.NetworkException;
        !DatM.cancel; !Card.revert;
        !Usr.NetworkException
      )
    )
    +
    ( // exceptions of Card.lookUp
      ?Card.NetworkException;
      !DatM.cancel; !Card.revert;
      !Usr.NetworkException
    )
  )
  +
  ( // exceptions of DatM.preReserve
    ?DatM.PreReservationException; !Log.exEvent; !Log.print;
    !Usr.buyTicket
  )
...
```

Obviously, a part of the complexity of the problem is the fact that we have to separate requests and responses of method calls to capture that exceptions can happen between them. Moreover the "reaction" inside such a call has to be divided into a "regular" and an exception part, and, even worth, the exception part has to contain repeatedly the "regular" continuation of the method (notice how

many times is `!Log.print` appears in the specification). Clearly, if we could take advantage of keeping the expressive power of the abbreviations `?a{P}` or `!a{P}`, and add specific syntactical constructs for capturing exceptions as classical programming languages do, we could shorten this behavior protocol significantly and make it much more concise and, consequently, easier to comprehend.

Another option is to use the *approximation by alternative* technique the basic idea of which is to put after any method call alternative, non-deterministically chosen reactions (`+`) covering all the potential continuations. These include "regular" continuation, and those specific for each of the exceptions the method can throw. An example of this technique is below. For instance, `!DatM.reserve` is followed by alternatively calling `!DatM.commit` or handling the reservation exception (the `!DatM.cancel; !Card.revert; !Log.exEvent; !Log.print` part). Obviously, this approach only approximates real behavior of a component by not explicitly specifying the issuing and accepting events related to an exception.

Technique: Approximation by alternative

```
...
Usr.buyTicket {
  !DatM.preReserve;
  ( !Card.lookUp;
    ( !Card.payment;
      ( !DatM.reserve;
        ( !DatM.commit;
          (
            null +
            // exception on DatM.commit
            (!DatM.cancel; !Card.revert; !Log.print)
          )
        )
        +
        // exception on DatM.reserve
        (!DatM.cancel; !Card.revert; !Log.exEvent; !Log.print)
      )
      +
      // exception on Card.payment
      (!DatM.cancel; !Card.revert)
    )
    +
    // exception on Card.lookUp
    (!DatM.cancel; !Card.revert)
  )
  +
  // exception on DatM.preReserve
  (!Log.exEvent; !Log.print)
}
...
```

3.2 Analyzing the problem and sketching a solution

In this section, we discuss all the key aspects related to expressing/capturing exceptions and behavior protocols at the level of an ADL (Architecture Description Language). In our view, the driving facts are:

1. In ADLs, exceptions should be specified with a granularity of a method (most likely in the interface specifications).
2. In ADLs, the key abstractions the protocols are associated with are frame protocols.
3. Issuing a method call in a frame protocol means the call goes outside of the component.
4. Throwing an exception in a method means an abnormal end of the method call.
5. Because of (3) an exception has to be handled in the frame protocol of the component which issued the call, and, because of (4), such handling is a specific reaction of the calling component after receiving the exception. In principle, this reaction has to be reflected by an adequate "traffic" on the calling component's interfaces.

Obviously an abnormal end of a method `i2.m` call from interface `i` can be easily modeled by replacing the standard "end_of_call" response `!i.m` by an exception response, e.g. `!i.e`. Moreover, the "abnormality" has to be reflected by abandoning the original protocol specifying the execution of m, i.e. the action `!i.e` in the protocol `P` appearing in the context `?i2.m{P}` has to be the last action generated by `P`. However, as the example in Sect. 3.1 indicates, addressing these abnormalities by the standard behavior protocols means becomes cumbersome. Since any protocol can be interpreted as an abstraction of code, we can, for this purpose, advantageously adopt a Java inspired construct of the form `?i2.m{... throw !e ...}` with the meaning (informally put) `throw !e` generates `!i.e` and then the execution of `i2.m` internals directly jumps to the lexically nearest }. In a similar vein, for handling an exception in a caller's frame protocol ((5)), we can adopt a `try {P} catch {?i.e:Q}` construct with the meaning very similar to the interrupt operator in CSP (i.e. $\texttt{P}\triangle_i\texttt{Q}$): if the event at the beginning of `Q` occurs, then the execution of the process `P` is abandoned and the process `Q` executes further. Along these lines, the event `?i.e` is the first one generated by the `catch {?i.e:Q}` construct.

However, we have to analyze exception throwing, propagating, and handling in all the (1)..(4) contexts below, since the methods are called across component boundaries and components can be nested. These four contexts represent all the situations on interface bindings related to a method call and an exception throwing and handling. These are client (1) and server (2) positions at a binding when no nesting is considered and the related situation when component nesting is taken into account. The latter are: nested server (3 - delegation) and nested client (4 - subsumption) positions at a binding.

(1) Client position (Figure 2)

Consider the component `X` which calls the method `a` on the interface `A`. If an exception `e` can be thrown by the call of `a` (i.e. thrown by `Y` in the setting of Fig. 2), the construct `try {... !A.a ...} catch {?A.e: ...}` is to be used in the frame protocol of `X` in order to handle the exception. An unhandled exception would cause an error (bad activity in terms of [1]).

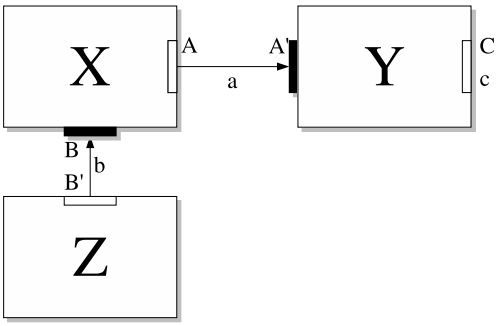


Fig. 2. Client-server

(2) Server position (Figure 2)

Consider the component `Y` accepting a call of `a` through the interface `A'`. In general, as mentioned above in this section, an exception in the execution of a is expressed by:

```
?A.a { ... throw !e; ... }
```

Based on the experience with our case studies, typical special cases of throwing an exception are:

1. An exception `e` is thrown due to an invalid actual parameter of `a` (invisible in protocols, but important for a credible abstraction). In protocols, this is typically expressed as
   ```
   ?A.a { null + throw !e; ... }
   ```
2. An exception is thrown due to a faulty return value in a nested call `!C.c`: (again invisible in protocols, but important for a credible abstraction). In protocols this is typically expressed as
   ```
   ?A.a { ... !C.c; (null + throw !e); ... }
   ```
   Since both in (1) and (2) the exception is a reaction on an "invisible" invalid value, it is a good practice to indicate the fact by choosing a mnemotechnical name for the exception (in Sect. 4, there are several examples of this method).
3. An exception is thrown in a `catch` construct. This is typical for exception propagation (even under a different name). For example, in
   ```
   ?A.a { ... try {!C.c} catch {?C.e1: throw !e2); ... }
   ```
   the `C.c` method can throw an `e1` exception, which is then converted into `e2`.

(3) Delegation (Figure 3)

Delegation basically means forwarding an acceptance of a call to an internal component[15]; in Fig. 3 the component `Y` delegates calls from the component `X` on the interface `A` to the interface `A'` in the internal component `ZA`. In principle, an exception `e` thrown in `ZA` in its method `a`, has to be delivered to the original caller, i.e. to the component `X`. Since the internals of `Y` are not visible to `X`, throwing of `e` should be specified not only in the frame protocol of `ZA` and but also of `Y`. Notice, however, that an exception thrown by the component `ZC` and handled by the component `ZA` would not be visible in the frame protocol of `Y`.

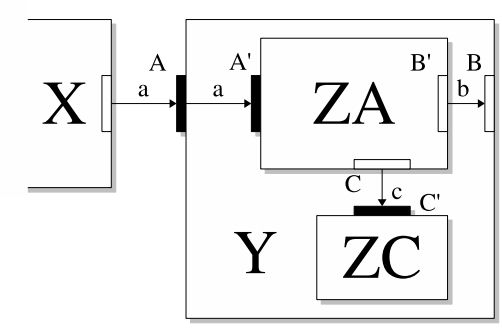


Fig. 3. Delegation

(4) Subsumption (Figure 4)

Subsumption basically means forwarding a call issued in an internal component to its parent component[15]; in Fig. 4, the component `ZA` subsumes the calls on the interface `A'` to the interface `A` in its parent component `X`.

Apparently, an exception thrown in `Y` is to be delivered to and handled by the caller, i.e. the component `ZA`. However, the component `X` is also in the client position with respect to `Y` (and, at a design stage, the internals of `X` do not have to be known). Therefore, handling of the exception has to be specified also in the frame protocol of `X`.

Since handling an exception in the frame protocol of `X` in general causes a "recovery communication" of `X` visible outside of `X`, potentially including a specific communication on its interface `B`. Obviously, this recovery communication should be adequately captured in the architecture protocol of `ZA` and `ZB` and, in particular, triggered by handling the exception in the frame protocol of `ZA`.

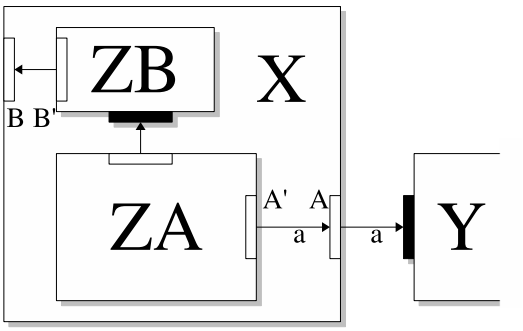


Fig. 4. Subsumption

3.3 Proposed solution - details

The main purpose of this section is to describe in more detail the semantics of the constructs introduced in Sect. 3.2 and to analyze the influence of these protocol enhancements on protocol compliance evaluation[1,3,15]. By convention, we will refer to exception handling based on these constructs as *Explicit try-catch* technique.

Throwing an exception

Syntax: `throw !exception_name`

This construct has to appear only in a protocol `P` written in the context of the form `i.a{P}`, i.e. inside the curly brackets abbreviation expressing call acceptance of a method `a`. In principle, `throw !exception_name` means that in the resulting trace the event `!i.a` modeling return from `a` is replaced by the event `!i.exception_name` and, at the same time, this is the last event generated by `P`. Should `P` contain nested method accepting constructs (such as `?i.b{Q}`), this principle is applied recurently.

For example, `?i.m{!a.x;?a.sl; throw !ex;!a.y}` would always generate the trace `?i.m,!a.x,?a.sl,!a.ex`. In a similar vein, `?i.m{X;(null + throw !ex); Y}` is equivalent to `?i.m; X;(!i.ex + Y; !i.m)` for some protocols `X` and `Y`.

It should be emphasized that `!exception_name` is always the last event generated by `P`, even though `P` contains a `|` and/or `||` operator. For example, the traces generated by `?i.m{(!a.x;?a.sl; throw !ex;!a.y)||!a.z*}` include (the beginning resp. end of a trace is denoted by `<` resp. `>`):

```
<?i.m;!a.x;!a.x;?a.sl;!a.sl;!i.ex>
<?i.m;!a.z;!a.x;?a.sl;!a.z;!a.z;!i.ex>
<?i.m;!a.x;!a.z;!a.z;?a.sl;!a.z;!a.z;!a.z;!a.ex>
<?i.m;!a.x;!a.z;?a.sl;?a.z;!a.sl;!a.z;!a.ex>
```

Anyhow, the reason why `throw !exception_name` generates the last event in `P`, no matter how many parallel activities in `P` are specified, is that it is hard to define a "reasonable" semantics of more than one exception (the remaining parallel activities could also throw an exception). As an aside, by opting for "interrupting" all the parallel activities we basically follow the semantics chosen in CSP for the interrupt operator[10].

Catching an exception

Syntax:

`try` { A }
`catch` {?$\mathtt{i}_{1,1}$.$\mathtt{exception\_name}_{1,1}$ ..., ?$\mathtt{i}_{1,m_1}$.$\mathtt{exception\_name}_{1,m_1}$: B_1}
`catch` {?$\mathtt{i}_{2,1}$.$\mathtt{exception\_name}_{2,1}$ ..., ?$\mathtt{i}_{2,m_2}$.$\mathtt{exception\_name}_{2,m_2}$: B_2}
...
`catch` {?$\mathtt{i}_{n,1}$.$\mathtt{exception\_name}_{n,1}$..., ?$\mathtt{i}_{n,m_n}$.$\mathtt{exception\_name}_{n,m_n}$: B_n}

where A, B_j are protocols and $\mathtt{i}_{ij}$ are interfaces. If a `throw !exceptionname`$_{ij}$ is applied in A in a context

`try { ... !A.a ...} catch {?`$\mathtt{i}_{ij}$`.exception_name`$_j$`: ...:`B_i`}`,

then the next event generated by the `try` construct is the first event specified by B_i. For simplicity, all the exceptions which could be thrown in the `try` construct have to be listed exactly once in one of the `catch` parts of the construct.

Influence on compliance evaluation. The exception-related constructs preserve the semantics of the operators defined for behavior protocols (in particular the semantics of the composition and consent operators important for composition error detection and compliance evaluation[1,15]).

Unhandled (uncaught) exceptions are captured statically by the protocol checker (information about the possible exceptions have to be a part of interface specification in ADL). An improper/non-existent reaction to an exceptions is typically captured as a bad activity error.

It can be easily shown that these constructs are without difficulty captured by the LTS representing a behavior protocol. For example, the LTS representing

```
try { !i.a1; !i.a2; !i.a3}
catch { ?i.e1: !i.b}
catch { ?i.e2: !i.c}
```

can be easily constructed by adding transitions to the states representing the methods' calls in which `e1` and `e2` can be returned: these transitions will lead to the LTS representation of the `e1` and `e2` handlers (Fig. 5). Since the resulting LTS remains a finite automaton, the finite trace-based semantics of the behavior protocol operators is preserved.

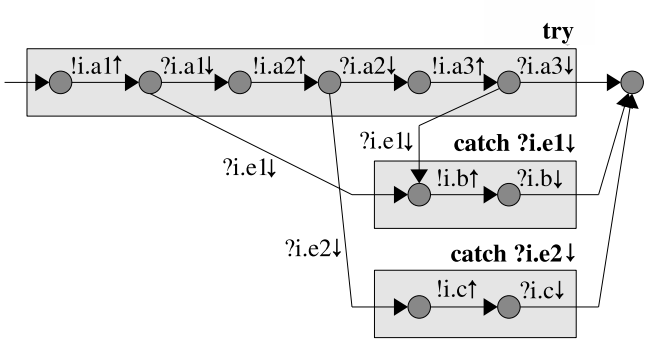


Fig. 5. Transition diagram

4 Case study

In this section, using again the example from Fig. 1, we illustrate how the exception-related constructs simplify specification of exceptions in the behavior protocols of the components introduced in Sect. 2.

The frame protocol of the Ticket manager is shown below. The component was already described in the Sect. 2; here we present a slightly more detailed version which includes the initialization of other components and the check of the `lookUp` method return value.

Ticket manager frame protocol

```
?Usr.init { !Card.init; !DatM.init};
(
  ?Usr.buyTicket {
    try {
      !DatM.preReserve; !Card.lookUp;
      null + (!Card.payment; !DatM.reserve; !DatM.commit)
    }
    catch { ?DatM.PreReservationException:
      !Log.exEvent;}
    catch { ?Card.NetworkException:
      !DatM.cancel; !Card.revert;
      throw !NetworkException}
    catch { ?DatM.DatabaseException, ?DatM.ReservationException:
        !DatM.cancel; !Card.revert; !Log.exEvent };
    !Log.print;
  }
  +
  ?Usr.returnTicket {
    try { !Card.revert; !DatM.free; !DatM.commit }
    catch {?DatM.DatabaseException:
      !DatM.cancel; !Log.print}
  }
)*;
?Usr.finish {!DatM.finish; !Card.finish}
```

Below is the frame protocol of the VISA component. After being initialized by accepting an `init` call, the "business" stage takes place: `lookUp`, `payment`, and `revert`. The `lookUp` and `payment` methods may throw an exception due to the problem on the network (`null + throw !NetworkException`). The `lookUp` method also verifies the card number via the verify method on the Blacklist interface. If the verification yields a negative result, `ListException` is thrown and validity is re-checked by the operator (call of `askValidity` on the Operator interface).

VISA frame protocol

```
?Card.init;
(
  ?Card.lookUp {
    try { !Blacklist.verify }
    catch { ?BlackList.ListException: !Oper.askValidity};
    null + throw !NetworkException
  }
  +
  ?Card.payment {
    null + throw !NetworkException}
  +
  ?Card.revert
  )*;
?Card.finish
```

In a similar vein, the frame protocol of Database manager indicates that the `preReserve`, `reserve` and `commit` methods can be alternatively called after initialization. All of them communicate with the Storage component via a `!Strg.Access` call which can return `StorageException`. Notice that this exception is converted to the `PreReservationException` resp. `ReservationException` consequently delivered to the caller of `preReserve` resp. of `reserve` or `commit`.

Database manager frame protocol

```
?DatM.init { !Strg.init };
(
  ?DatM.preReserve {
    try { !Strg.Access* }
    catch { ?Strg.StorageException:
      throw !PreReservationExcpetion}
  }
  +
  ?DatM.reserve {
    try { !Strg.Access* }
    catch { ?Strg.StorageException:
      throw !ReservationException}
  }
  +
  ?DatM.commit {
    try { !Strg.Access* }
    catch { ?Strg.StorageException:
      throw !ReservationException}
  }
  +
  ?DatM.cancel
)*;
?DatM.finish { !Strg.finish }
```

Storage frame protocol

```
?Strg.init;
(
  ?Strg.access { null + throw !StorageException}
)*;
?DatM.finish
```

The frame protocol of Reservation describes the communication with the environment of the whole reservation application. Notice that the exceptions which are thrown and handled inside the component are naturally not visible at this level, but `NetworkException` is propagated through the Reserv interface so that it has to appear in the frame protocol in the `throw` construct. On the other hand, `ListException` is handled in this frame protocol as `null}` since its handling does not require external component communication (as an aside, details of its handling are visible the VISA frame protocol - `!BlackList.test` is subsumed from VISA). In contrast, if a OperatorVerification component were outside Reservation (Fig. 6), details of `ListException` handling would be visible in the Reservation frame protocol, as illustrated in it by the comment line.

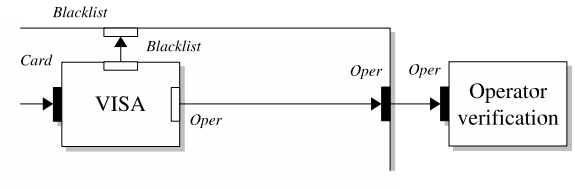


Fig. 6. Modified reservation component

Reservation frame protocol

```
?Reserv.init;
(
  ?Reserv.buyTicket {
    !Log.exEvent;
    !Log.print
    +
    (
      try { !BlackList.test }
      catch { ?BlackList.ListException: null};
                // catch { ?BlackList.ListException: !Oper.askValidity}
      (!Log.exEvent + null);
      !Log.print + throw !NetworkException
    )
  }
  +
  ?Reserv.returnTicket
)*;
?Reserv.finish
```

5 Evaluation

This work was inspired by our experience gained during our attempt to a apply behavior protocols to a non-trivial, real-life component-based application. We had chosen the Speedo project[22] available from the ObjectWeb consortium as an open source implementation of the Sun JDO specification[23]. The implementation is based on the FRACTAL component model[7] and is heavily using the Perseus persistence framework[14]. Together, behavior protocols of 26 components were written. Our experiences has been that without an explicit notation for exception handling, protocols are very hard to read and comprehend, and furthermore, the correspondence between the behavior specification and code is very hard to trace. We support this claim be the figures provided in Fig. 7. Here, the length of behavior protocol specification is given for four specific techniques of expressing exceptions via behavior protocols. The "Ignoring exceptions" techniques means specifying behavior in a way which does not consider exceptions at all. The "Explicit try-catch" technique is based on the behavior protocol extensions described in Sect. 3.2, while "Intrinsic exceptions" and "Approximation by alternative" are the methods described in Sect. 3.1.

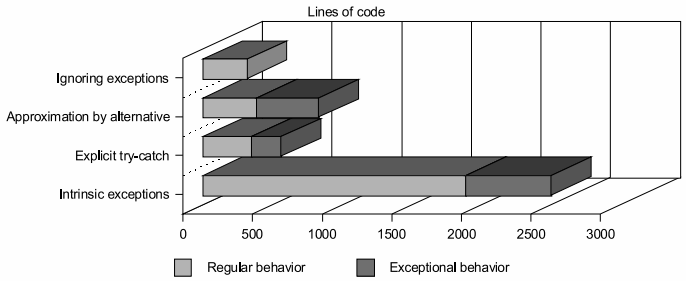


Fig. 7. Description complexity of the Speedo project formalism

Each bar of the graph is divided into two parts to indicate the number of lines specifying the "regular" behavior (gray) and exception-related behavior (black). From the chart it is clearly visible how significantly the proposed "Explicit try−catch" construct shortens behavior specification. Both "Approximation by alternative" and "Explicit try−catch" do not cause any significant grows of the "regular" part of the behavior specification, in contrast to the "Intrinsic exceptions" technique where often some of the specification sections have to be repeated. Notice also that "Approximation by alternative" causes grows of the exception-related behavior specification in comparison with "Explicit try−catch".

6 Related work

There are many publications on exception handling, however not many of them are related to exceptions at a level of abstraction higher than source code.

In [20] the authors employ the C2 architectural style featuring composition contracts. Components have top and bottom interfaces connected via connectors responsible for routing and filtering asynchronous messages. There are two types of messages - a request message and a notification message depending on whether the message flows up or down though the system. This is very similar to our request-response notation. The composite contract (a service-implementing component) ends either with a normal notification or an exceptional notification. In the latter case, an exception handler component is activated. If the exception recovery is successful, an abort notification is generated; otherwise a failure notification is generated and the component may be left in an inconsistent state. In this approach, the contract component "remotely" corresponds to our `try` construct and exception component to the `catch` construct, however, the philosophy of component hierarchy is different compared to ours and there is no behavior specification at the level of the whole component.

The static source analyzing tool PREfast[13] checks all the execution traces for possible erroneous behavior (typically null reference, memory leaks). In the context of this paper, it is interesting that during exception propagation some (predefined) functional failures are detected, such as missing memory deallocation and resource unlocking. This property is checked by our approach implicitly - communication errors would be detected in the behavior composition process[3].

Session types are used for describing behavior of CORBA IDL in [24]. The approach of behavior description is similar to our interface protocol (protocols restricted to an

interface), with a different syntax though. An exception is expressed in the specification of potential responses of a method. However, if the method can raise more exceptions, the same label is used for each of them.

A CSP based exception handling is introduced in [9]. The exception operator ($\overrightarrow{\triangle}$), is inspired by the CSP interrupt operator $\triangle_i$[10]. While $\texttt{P}\triangle_i\texttt{Q}$ means preemption of `P` on an externally coming event `i` and continuation by `Q` (`i` is the first event of `Q`), the exception operator considers in $\texttt{P}\overrightarrow{\triangle}\texttt{Q}$ the event `i` as an internal event and therefore `Q` can be interpreted as an exception handler and `P` as a `try` construct. Since in our proposed extension of behavior protocols the composition of two components' behavior also yields an internal action τ`e` (one of the components throws an exception `!e` and another one accepts it via `?e` in a `catch` construct), the approaches are similar in this respect. However, there are significant differences. In CSP, interrupts can occur without an intervention of the original process `P`, thus being similar to hardware interrupts. In our approach, an exception is triggered by invoking a method call and it has to be an expected event. Additionally, one catch block can handle more that one exception to avoid repeating of the same handling routine if an identical reaction is desirable. Also exception handling can be subject of compliance tests of both horizontal and vertical contracts (Sect. 2).

7 Conclusion

The key contributions of this paper include:

(i) An analysis of the role and importance of exceptions in behavior specification of software components is given and it is shown how behavior protocols can be extended to handle exceptions in an efficient way in terms of readability, comprehension, and the size of a behavior specification.

(ii) This claim is supported by providing experimental results from a real-life case study of applying different exception handling techniques based on behavior protocols. From these experiments, it is clearly visible how significantly the proposed behavior protocol extension by an explicit exception handling construct shortens the behavior specification of a non-trivial component-based application.

Acknowledgments

We would like to give a special credit to our colleagues Jiri Adamek and Vladimir Mencl for their valuable comments, and to Jan Kofron and Pavel Jezek for another non-trivial case study and a number of suggestions.

References

1. J. Adamek, F. Plasil: Component Composition Errors and Update Atomicity: Static Analysis, Journal of Software Maintenance and Evolution: Research and Practice 17(5), pp. 363-377, 2005 2, 7, 10, 11
2. J. Adamek, F. Plasil: Erroneous Architecture is a Relative Concept, in Proceedings of SEA conference, Cambridge, MA, USA, ACTA Press 2004 2, 3
3. J. Adamek, F. Plasil: Partial Bindings of Components − any Harm?, in the Proceedings of APSEC 2004, IEEE Computer Society, pp. 632−639, Nov 2004 2, 10, 15

4. L. de Alfaro, T. A. Henzinger: Interface Automata, in Proceedings of the 9^th Annual ACM Symposium on Foundations of Software Engineering (FSE), 2001 1
5. R.J. Allen, D. Garlan: A Formal Basics For Architectural Connection, ACM Transactions on Software Engineering and Methology, Jul 1997 1
6. W. DePrince jr., C. Hofmeister: Enforcing a lips Usage Policy for CORBA Components, in proceedings of EUROMICRO'03, Sep 2003 1
7. Fractal component model: http://fractal.objectweb.org/ 14
8. D. Harel: Statecharts: A Visual Formalism for Complex Systems, Science of Computer Programming 8, Elsevier Science Publishers B.V., 1987 1
9. G.H. Hilderink: Managing Complexity of Control Software through Concurrency, PhD thesis, University of Twente, The Netherlands, ISBN 90-365-2204-8, May 2005 16
10. C.A.R. Hoare: Communicating Sequential Processes, Prentice-Hall International, UK, Ltd., ISBN 0-13-153271-5, 1985 10, 16
11. J. Magee, J. Kramer: Concurrency: State models & Java programs, John Wiley & Sons Ltd, ISBN 0-471-98710-7, 1999 1
12. Object Management Group: UML 2.0 Infrastructure Final Adopted Specification, OMG document ptc/03-09-15, Sep 2003 1
13. PREfast: http://www.microsoft.com/whdc/devtools/tools/PREfast.mspx 15
14. Perseus persistence framework: http://perseus.objectweb.org/ 14
15. F. Plasil: Enhancing Component Specification by Behavior Description – the SOFA Experience, in Proceedings of the 4th WISICT 2005, A volume in the ACM, Computer Science Press, Trinity College Dublin, Ireland, pp. 185−190, Jan 2005 8, 9, 10, 11
16. F. Plasil, S. Visnovsky: Behavior Protocols for Software Components, IEEE Transactions on Software Engineering, vol. 28, no. 11, Nov 2002 1, 2, 3, 4
17. F. Plasil, S. Visnovsky, M. Besta: Bounding Behavior via Protocols, in Proceedings of TOOLS USA '99, 1999 3
18. R.H. Reussner, S. Becker, V. Firus: Component Composition with Parametric Contracts, Tagungsband der Net.ObjectDays, 2004 1
19. H.W. Schmidt, B. J. Kramer, I. Poernomo, R. Reussner: Predictable Component Architectures Using Dependent Finite State Machines, Proceedings of the 9th International Workshop in Radical Innovations of Software and Systems Engineering in the Future, LNCS Springer-Verlag, ISBN 3-540-21179-9, Vol 2941, 2004 1
20. R.M. Silva, P.A.C. Guerra, C.M.F. Rubira: Component Integration using Compositions Contracts with Exception Handling, in Proceedings of ECOOP2003 Workshop on Exception Handling in Object-Oriented Systems, TR 03-028 Univ. of Minnesota, Dept. of Comp.Sci., Jul 2003 15
21. SOFA project: http://sofa.objectweb.org/, http://nenya.ms.mff.cuni.cz/projects.phtml?p=sofa\&q=0 3
22. Speedo: http://speedo.objectweb.org/ 14
23. Sun JDO specification: http://java.sun.com/products/jdo/ 14
24. A. Vallecillo, V.T. Vasconcelos, A. Ravara: Typing the Behavior of Objects and Components using Session Types, 1st International Workshop on Foundations of Coordination Languages and Software Architectures. Electronic Notes in Theoretical Computer Science, 2002 15
25. K. Wallnau: Volume III: A Technology for Predictable Assembly from Certifiable Components, Technical Report CMU/SEI-2003-TR-009, Apr 2003 1

Chapter 6

Reducing Component Systems' Behavior Specification

Viliam Holub and František Plášil

Contributed paper at **XXVI International Conference of the Chilean Computer Science Society** [3].

In conference proceedings,
accepted for publication by IEEE Computer Society,

Reducing Component Systems' Behavior Specification

Viliam Holub
Distributed Systems Research Group
Charles University, Czech Republic
Email: holub@dsrg.mff.cuni.cz

Frantisek Plasil
Institute of Computer Science
Academy of Sciences of the Czech Republic
Email: plasil@cs.cas.cz, plasil@dsrg.mff.cuni.cz

***Abstract*—Behavior verification of large component systems suffers of state explosion in particular when components involve parallel activities. For behavior protocols, a method of component behavior specification, we present a method of state space size reduction based on symbolic manipulation with the specification done by applying a set of reduction rules. A case study is presented showing that the specification size is often reduced to only a fraction of the original one.**

I. Introduction

A. Behavior protocols

Behavior protocols [1] were designed as a specific process algebra, to specify the desired finite sequences of method calls on component interfaces (their interplay) - the behavior of components.

A behavior protocol P is an expression that generates a set of traces of event tokens representing atomic events (actions) related to method invocations ($?m\uparrow$ stands for accepting a method m invocation, $!m\uparrow$ issuing an invocation of m, $?m\downarrow$ accepting the response (end) of m's execution, $!m\downarrow$ issuing the response). In addition to event token, P is composed of operators (';' sequencing, '$+$' alternative, '$*$' repetition, and '$|$' parallel interleaving without a communication), and abbreviations $?m$ (stands for $?m\uparrow$; $!m\downarrow$), $?m\{P\}$ (meaning $?m\uparrow; P; !m\downarrow$); similar rules are introduced for $!m$, and $!m\{P\}$.

The behavior protocol specifying the behavior of a particular component is called a *frame protocol*. As an example, consider the frame protocol of the AccountDatabase component depicted in Fig. 15 from Fig. 1. The protocol specifies that on its provided interface IAccount, it accepts a call of GenerateRandomAccountId, or alternatively ($+$), calls of CreateAccount and RechargeAccount. In the latter case as a reaction on accepting the call it issues a call of Withdraw on its required interface ICardCenter. This can be repeated a finite number of times ($*$). In parallel ($|$) to this, AccountDatabase can repeatedly accept calls on IAccount of AdjustAccountPrepaidTime_1, AdjustAccountPrepaidTime_2, and AdjustAccountPrepaidTime_3. Each time these adjusting calls can be accepted in a sequence a finite number of times.

Behavior protocols introduce special case of parallel composition (the consent operator $\bigtriangledown$) [2] with communication and hiding as known from the process algebra ACP [3]. It produces interleaving of events, while merging the invoke "!" and accept "?" events with the same name into an internal event "τ" which (similarly to CCS [4]) in principle means combining communication and hiding as defined in ACP. Moreover, accept events are blocking, while invoke events have to be merged by a counterpart immediately; unlike other process algebras, the consent operator produces specific event tokens corresponding to *composition errors* which are

- *bad activity* occurring when the issued event cannot be accepted,
- *no activity* (deadlock) when only accept events are enabled, and, since only finite traces are allowed,
- *divergence* (infinite activity) when the composition would produce an infinite trace.

By convention, a communication error is expressed by an error event token $!\epsilon$ for bad activity, $\oslash\epsilon$ for no activity, and $\infty\epsilon$ for infinite activity, which is always the final token in an erroneous trace [2].

B. Checking compliance

Because of its ability to identify communication errors, the consent operator is advantageously used to verify component behavior compliance. By composing the frame protocols of the communicating components on the same level of nesting (e.g. the frame protocols of ValidityChecker, CustomToken, and Timer in Fig. 1), it is verified that these components will cooperate correctly — *horizontal compliance* is verified. Naturally, this is true provided their implementation obeys their frame protocols.

In a similar vein, it is important to verify whether the composed behavior of the components cooperating at a particular level of nesting complies with the behavior specified for the surrounding (parent) component (Token in the example above). This *vertical compliance* is again verified with the help of the consent operator via the following trick: Even though the parent component in principle just mediates the calls (both incoming and outcoming) for its subcomponents, it can be easily turned into an 'environment' component which, instead of mediating, really issues and accept these calls. Its frame protocol is easily composed as the inverted frame protocol of the parent component, with ! replaced by ? and vice versa.

Going back to the example, the composition FPValidityChecker $\bigtriangledown$ FPCustomToken $\bigtriangledown$ FPTimer thus verifies the horizontal compliance and FPToken^{-1} $\bigtriangledown$ (FPValidityChecker $\bigtriangledown$ FPCustomToken $\bigtriangledown$ FPTimer) the vertical compliance. Here

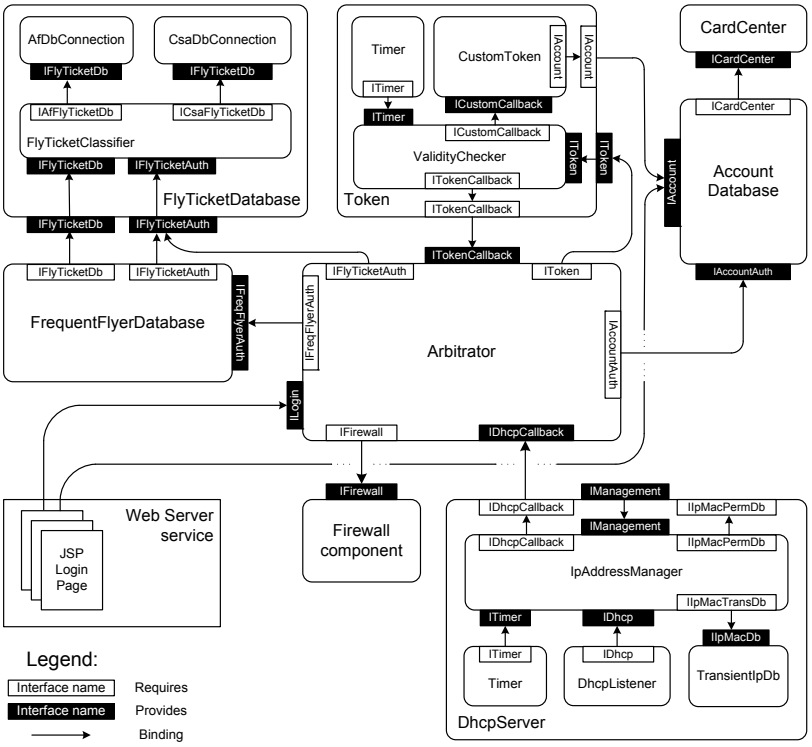


Fig. 1. Architecture of the Airport Internet providing service

FP stands for a frame protocol and $^{-1}$ denotes protocol inversion.

Verification of compliance is done by model checking — a number of specialized model checkers have been designed for this purpose [5]–[7]. Since parallel composition is involved via the operators $|$ and $\bigtriangledown$, the Cartesian products of the state spaces associated with the operands of each of these operators tend to run into the *state explosion problem*.

C. Goals and basic idea of contribution

State explosion is a problem inherent to model checking involving parallel activities. The typical techniques to address it include abstraction [8], abstract interpretation [9], and partial order reduction [10], [11]. The former is hard to apply to behavior protocols' compliance verification, since the level of abstraction at with they capture behavior of software components is already very high (they abstract from component state and method parameters). At a first sight, partial order reduction is much more promising, since (i) the events in operands of the $|$ operator do not communicate, however, their interleavings can significantly influence whether a non-blocking transition is enabled or not.

The goal of this paper is to show that state explosion in behavior compliance verification can be addressed by reducing the frame protocols via a technique which, in addition to (i), employs observation that (ii) each pair of components communicates by events with unique, dedicated names (composed of the name of an interface and method). Advantageously, this can be employed to predict the communication leading to τ actions in the composition done by the consent operator. The reduction technique, reduction process (Sect. II), is based on a set of heuristic reduction rules (Sect. III). In some cases, the reduction can even eliminate the need of actual model checking, as the frame protocols involved in parallel composition get reduced to $NULL$. Since the original behavior specification (frame protocols) is modified, it is very important to find a way to interpret a counter example found by a model checked in the original frame protocols. This is shown in Sect. IV. The proposed reduction process was applied in a case study (Sect. V) with a positive experience (Sect. VI).

II. Reduction process

A. Overall strategy

The classical method of system verification via model checking consists of three steps. First, the user writes a specification (a model) of the system. Then, the specification is verified by a model checker. Finally, the result of the verification (report on correctness or a counter example) is

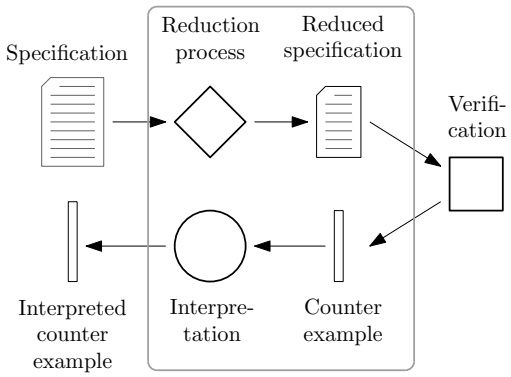


Fig. 2. Overall strategy

presented to the user. We add two more steps: reduction and counter example back interpretation (Fig. 2).

a) Reduction: Before running the model checker, the specification is analyzed and reduced (by a tool) in order to lower the size of the corresponding state space. In addition to the reduced specification, the tool produces also a log of the modifications done in the original specification. The reduced specification is then passed to the model checker.

A reduction requires a thorough knowledge of the relations among different concurrent activities involved in a consent operation. Therefore, with the aim to efficiently capture these relations, a frame protocol is represented as a hierarchy of LTSs, following the syntactical nesting of the | operators; roughly speaking, a frame protocol is represented as a hierarchy of parallel automata. Moreover, to capture potential communication in the consent operation, *counterpart* relation is maintained in addition. The basic idea is that pairs of the form ($!i.a\uparrow$, $?i.a\uparrow$) and ($!i.a\downarrow$, $?i.a\downarrow$) are in the relation. The conversion from textual frame protocols to this LTS internal representation is described in Sect. II-B.

The actual reduction is achieved by a repetitive application of reduction rules. Each reduction rule describes a frame protocol modification (its LTS modifications) and the set of conditions which must be satisfied in order to apply this rule. The list of reduction rules and the associated conditions are described in Sect. III.

b) Counter example back interpretation: As the model checker finishes by reporting an error in the reduced specification and not in the original one, the counter example has to be back interpreted for the user. This is achieved by applying inversion of all the modifications saved in the log; details are described in Sect. IV.

As the bottom line, the reduction is transparent to the user.

B. Converting Behavior protocol into LTS

As shown in Sect. I-A, the frame protocol of a component is an expression employing in addition to the classical regular expression operators $+$, ; and $*$ also | and $\triangledown$. While it is easy to convert a regular expression into LTS, when converting a frame protocol a special care must be taken for the parallel operators | and $\triangledown$. The semantics of $A|B$ is defined as all the possible interleavings of the traces generated by the protocols A and B. Obviously, creating a corresponding LTS based on this definition typically leads to enormous size of the LTS — it determines a subset of Cartesian product of the state spaces of A and B. Therefore, because the events in A and B do not communicate, they can be actually expressed as separate, parallel LTSs (basically following the idea of parallel automata). In general, in a frame protocol fp these LTSs can be nested, being constructed recursively by following the syntactic structure of fp.

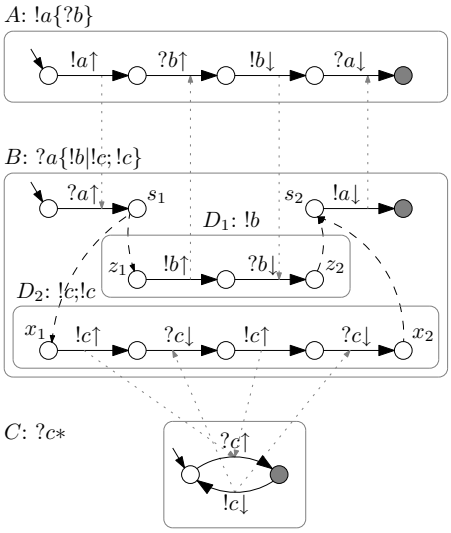


Fig. 3. The LTS representation of the parallel composition $!a\{?b\}\triangledown?a\{!b|!c;!c\}\triangledown?c*$

Consider an example of parallel composition of frame protocols of three components $!a\{?b\}\triangledown?a\{!b|!c;!c\}\triangledown?c*$, where the first component calls a method a and meanwhile accepts a callback b, the second component awaits a call of a and implements it as a parallel call of the methods b and c (two calls of the latter), and the third component accepts an arbitrary number of the c method calls. The corresponding LTSs working in parallel are depicted in Fig. 3.

We graphically capture LTSs as boxes with round corners (sometimes except for the topmost LTS). In the protocol $?a\{!b|!c;!c\}$ represented by the LTS B, the execution splits after $?a\uparrow$ from state s_1 in two nested LTSs D_1 and D_2, with the initial states z_1 and x_1. We say that states z_1 and x_1 are *associated* with s_1. By convention, associations are depicted by dashed arrows like those from the state s_1 to the states z_1 and x_1. In a similar vein, when both LTSs D_1 and D_2 finish in the states z_2 and x_2, the execution joins in the state s_2 and continues in the LTS B.

III. Reduction Rules

A. Elimination of ν-transitions

As mentioned in Sect. I-A, a τ event is created as a result of a parallel composition (via $\triangledown$) of an invoke and accept events with the same name; in some process algebras it is said that such two events *synchronize* or *communicate*. Assuming a τ event is produced in the result of $A \triangledown B$ by synchronizing $!a\uparrow$ and $?a\uparrow$, it becomes here an internal action which cannot synchronize any further. In particular, in a subsequent parallel composition such as $(A \triangledown B) \triangledown C$, this τ represents an "uninteresting" asynchronous action. However, its presence may be important to express that some other events are not immediately enabled; this holds for each of the parallel compositions $A \triangledown B$ and $(A \triangledown B) \triangledown C$. The key idea behind this paper is whether one could statically decide on eliminating the actions $!a\uparrow$ and $?a\downarrow$ directly from A resp. B (i.e. reduce A resp. B) for the purpose of compliance checking. Obviously, such an elimination should neither introduce a communication error, nor eradicate one. The challenge is to find the reduction rules which would guarantee this requirement. Since it is easier to articulate such rules for the LTS representation of a protocol, we will use this notation in the rest of this section.

Consider again the composition $A \triangledown B$ and event tokens $!a\uparrow$ resp. $?a\uparrow$ to appear in A resp. B. in such a way that they synchronize (which of the appearances can synchronize is determined from the counterpart relation).

To help articulate rules for making sure that elimination of such transition does not eradicate a communication error, consider first the following example:

$$(!a*|!b*) \triangledown (?a+?b)*$$

The corresponding LTS is in Fig. 4 where the abbreviations are expanded. By convention, the transition which are candidates for elimination are denoted as ν-transition — in this example these are the transitions originally labeled $?a\downarrow$ and $!a\downarrow$. Semantically, a ν-transition is an empty transition ($NULL$ in protocols) not visible in any trace. It is similar to the ϵ-transition in automata theory [12], where this transition accepts an empty input string. Notice, however, that if the ν-transition was eliminated in Fig. 4, the $\triangledown$ composition would eradicate the bad activity error present in the original composition $(!a*|!b*) \triangledown (?a+?b)*$. The error is caused by $!b\uparrow$ which cannot be accepted immediately when $!a\uparrow$ is accepted, since $!a\downarrow$ is to be issued first. The corresponding error trace is $\tau; !b\epsilon$. Obviously, the ν-transitions were not good candidates for elimination. Below are the rules for a save removal of a ν-transition.

Consider a ν-transition $\overrightarrow{s_1 s_2}$. If it complies with the following rules, it is guarantied that no communication error will be eradicated by $\overrightarrow{s_1 s_2}$ removal:

(i) If s_2 has only outgoing transitions and $s_1 \neq s_2$ (Fig. 5a), the transition is removed and states s_1 and s_2 are joined into a single state $s_1 \circ s_2$.

(ii) If s_2 features both outgoing and incoming transitions other than the ν one (Fig. 5b), the ν-transition $\overrightarrow{s_1 s_2}$ is removed and all the outgoing transitions from s_2 (such as δ) are duplicated by introducing them to s_1 as well.

(iii) If $s_1 = s_2$, the transition can be safely removed; this is a special case of (ii).

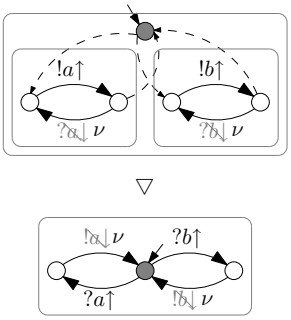


Fig. 4. Naive ν-transition elimination

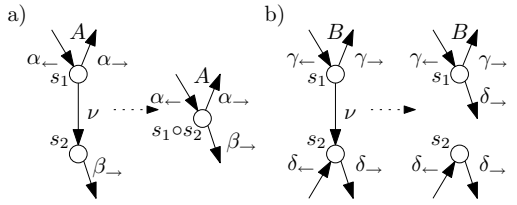


Fig. 5. Elimination of ν-transition

To justify the rules, assume a ν-transition $\overrightarrow{s_1 s_2}$ was eliminated and a bad activity error $!a\epsilon$ was eradicated by that. Then the result of the elimination accepts an event a which would not be accepted in the original LTS. Therefore there has to be a transition outgoing from s_2 accepting a, but none such transition outgoing from s_1. Hence to avoid this bad activity error, all the accepted events in s_2 must be accepted in s_1 as well.

B. Identification of ν-transactions

Consider a parallel composition of the form $A_1 \triangledown A_2$. This section provides a list of rules as to how to identify and eliminate ν-transitions from A_1 and A_2 in such a way that no communication error will be injected into this parallel composition. At the same time, applications of these rules assume that the conditions (i) - (iii) from Sect. III-A are satisfied. Again, the rules are articulated for the LTS representation of A_1 and A_2.

1) Simple method call: This rule (Fig. 6) addresses calls of a method a, assuming that no "reactions" via $\{\ldots\}$ for both accepting and issuing of such a call are specified in A_1 and A_2; i.e. each of the accepting call specification in A_2 takes the form $?a\uparrow; !a\downarrow$, and each issuing of such call in A_1 is specified as $!a\uparrow; ?a\downarrow$.

Obviously, the basic idea is that, with respect to the consent operator, the transitions $?a\downarrow$ and $!a\downarrow$ are "redundant"

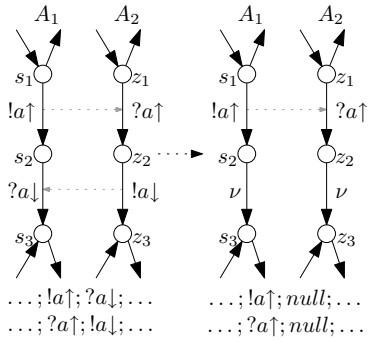


Fig. 6. Simple method call

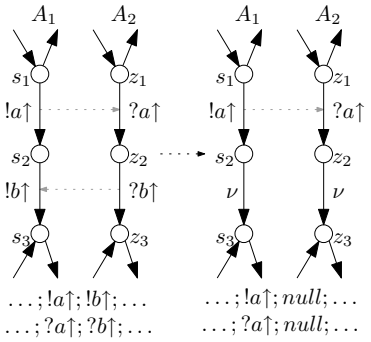


Fig. 7. Serial method invocation

in the specification, can be replaced by ν-transitions, and reduced. The correspondence of the state triples (s_1, s_2, s_3) and (z_1, z_2, z_3) is determined by the counterpart relation. This reduction does not introduce a communication error provided:

(i) there are no other transition from the states s_2 and z_2
(ii) for each instance of the state triple (s_1, s_2, s_3), there is an instance of (z_1, z_2, z_3) determined by the counterpart relation (and vice-versa)
(iii) the reduction is performed for all instances of (s_1, s_2, s_3) and (z_1, z_2, z_3) satisfying (ii).

To show that no communication error is injected into A_1 and A_2 by the application of this rule, assume no communication error is present in $A_1 \triangledown A_2$ due to the parallel composition of (s_1, s_2, s_3) and (z_1, z_2, z_3). Since (i) requires no other transition from the states s_2 and z_2 to exist, skipping of $?a\downarrow$ and $!a\downarrow$ - events that communicate, cannot introduce a communication error.

2) Serial method invocation: A sequence of method invocations (Fig. 7) is often a result of simple method call reductions. The basic idea is that since $!a\uparrow$ and $?a\uparrow$ synchronize, the following events $!b\uparrow$ and $?b\uparrow$ are "redundant" and can be safely replaced by ν-transitions and removed. Again, the correspondence of the state triples (s_1, s_2, s_3) and (z_1, z_2, z_3) is determined by the counterpart relation, and this reduction does not introduce a communication error provided:

(i) there are no other transition from the states s_2 and z_2,
(ii) for each instance of the triple (s_1, s_2, s_3), there is an instance of (z_1, z_2, z_3) determined by the counterpart relation (and vice-versa)
(iii) the reduction is performed for all instances of (s_1, s_2, s_3) and (z_1, z_2, z_3) which satisfy (ii).

Also to show that no communication error is injected into A_1 and A_2 by application of this rule, the same arguments as in the III-B1 hold.

3) Simple cycle: Let the specification of A_2 contain just a single state z and a transition t, labeled $?a\uparrow$ which thus begins and ends in z (Fig. 8a). Then all the events $!a\uparrow$ in A_1 which are in counterpart relation with t synchronize (unless they are unreachable in A_1). Therefore they (and t) can be safely replaced by ν transitions and removed. Since these events communicate while A_2 remains in state z, removing these ν transitions cannot inject a communication error. A similar rule can be articulated for a $!b\downarrow$ transition (Fig. 8b).

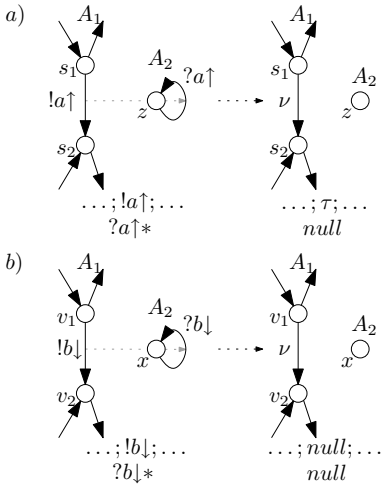


Fig. 8. Simple cycle

4) External events: Consider verification of vertical compliance of two components with the frame protocols A and B. The components do not communicate only with each other, but also with their "environment" via their interfaces not yet bound (there are external events in the frame protocols). Note, that these external events are unambiguously identified, since their names are unique (Sect. I-C). Assume now that there is an ideal environment with the protocol E. Here "ideal" means that it (i) accepts any external event issued by A and B, (ii) issues any event A and B are ready to accept as external, and (iii) does not issue any other external event. Then, obviously, $E \triangledown (A \triangledown B)$ can contain

only such communication errors which are caused by the communication of A and B. Therefore, all the transitions labeled by an external event in A and B can be safely replaced by ν-transitions and removed (Fig. 9).

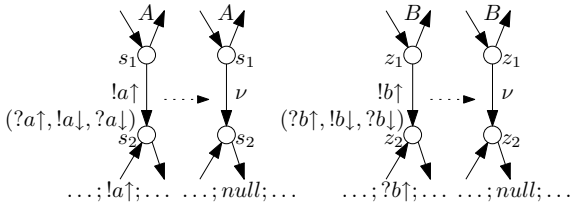


Fig. 9. External events

5) Initital state transitions: Consider again a parallel composition $A_1 \triangledown A_2$. If from the initial states of A_1 and A_2 there are only single transitions which synchronize (Fig. 10), they can be safely replaced by ν-transitions and removed provided there is no "third-party" synchronization in the state z_2 (or s_2) as depicted in Fig. 11. In this setting, the ν-transitions from the initial state of A_1 and A_2 would eliminate the bad activity error ($!b\downarrow$ cannot be accepted immediately in the initial state of A_2.)

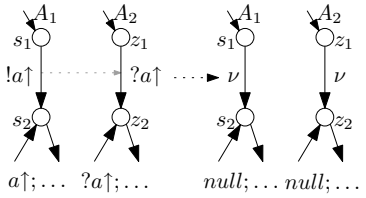


Fig. 10. Initial state transition

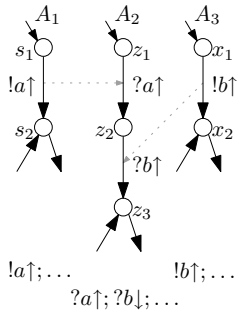


Fig. 11. Danger of hiding a bad activity error

C. Reduction outside $\triangledown$ composition

This section articulates three rules for reducing an LTSs in a "classical" automata minimalization way.

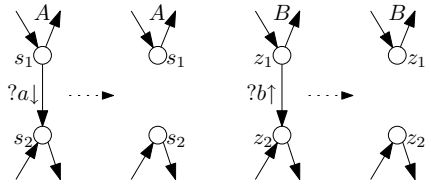


Fig. 12. Unbound actions

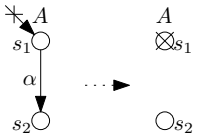


Fig. 13. Unreachable states

1) Unbound actions: Components often implement more business logic than required in a particular environment. Typically, the unused features are reflected in the component architecture as unbound interfaces. Obviously, in the frame protocol, no event at an unbound interface can be ever accepted, so that its acceptance never appears in a trace. Hence a transition labeled by such an accept event can be safely removed (Fig. 12). As an aside, on the contrary, issuing an event on unbound interface triggers a bad activity error in a consent composition.

2) Unreachable states: Obviously, unreachable states and consequently unreachable transitions are unnecessary in an LTS, since they do not contribute to any trace. At the same time, it is important to emphasize that reducing them may subsequently enable another reduction rules to be applied.

The basic idea of removing an unreachable states and all related transitions outgoing from it is illustrated in (Fig. 13). Here the state s_1 and also the transition α are removed in an LTS A. The unreachability of s_1 is emphasized by the crossed arrow. Unfortunately, to decide precisely whether a state is reachable (the reachability test) is time-consuming; typically it is as hard as the corresponding state space traversal itself. Therefore, for practical reasons, we impose a stronger condition in the reachability test: a state is unreachable if it is neither a starting state, nor an associated state, and has no incoming transition.

3) Redundant state: In general, by *redundant state* we mean a state which transitions (if any) do not contribute to any trace. However, for simplicity, we consider only four key situations when a redundant state is easily removable without introducing any communication error (Fig. 14). Even though the situations a) - d) in this figure look artificial, they typically result from a series of other reductions. Except for the trivial a) situation, the other reductions in Fig. 14 involve associate states related to nesting of LTSs in very special, pathologic situations, namely: A nested LTS contains just a single state and no transition (b), the outmost LTS contains just a single and associate state to initialize nested LTSs and another such

state for waiting the activities of these LTSs to be finished (c), and there is a redundant level of LTS nesting such as B in the case d).

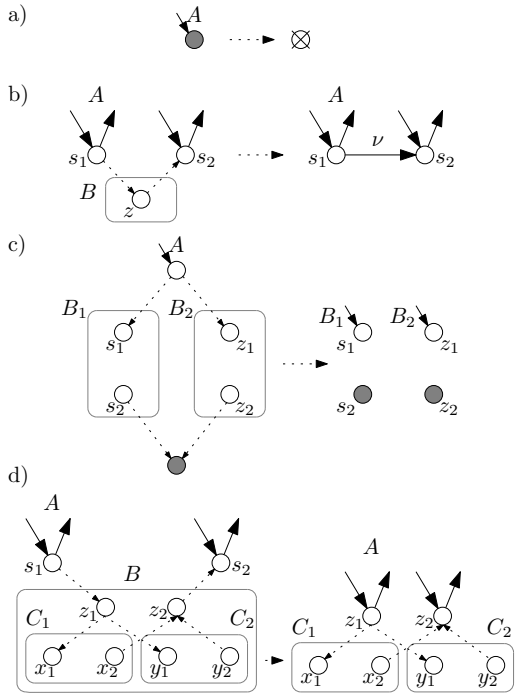


Fig. 14. Redundant state

IV. Counter example back interpretation

For every application of a reduction rule, we save in a log file all the states and actions (transitions) that have been affected by it. To back interpret the counter example, we take all the reductions from the log in reverse order and apply them inversely to the states and transitions in the counter example to enhance it accordingly.

The reductions B1, B2, B3, and B4 replace unnecessary transition by a ν transition, which is immediately reduced according to Fig. 5 as discussed in Sect III-A. The counter example must be modified when a ν transition was (virtually) executed. Although the execution of a ν transition cannot be explicitly expressed in the counter example, its effect is indirectly recoverable from the transition's starting state. In Fig. 5a, it is the transition from $\beta_{\rightarrow}$ starting at $s_1 \circ s_2$. In Fig. 5b, it is the transition from $\gamma_{\rightarrow}$ starting at s_1.

The reduction B5 represents a direct action execution (τ). To back interpret its effect, we have to explicitly add the reduced action at the beginning of the counter example.

The reductions C1 and C2 remove unemployed behavior and thus are not reflected in the counter example. The reduction C3 modifies internal structures only, so that it is not reflected in the counter example either.

TABLE I
Effects of reduction rules on the state space size

Step	Reduction	State space size	Time
0	Original specification	871122	143m42s
1	External events (26x)	36369	4m43s
2	Simple method call (10x)	9506	1m9s
3	Initial state transition(1x)	9504	1m4s
4	Redundant state (4x)	9504	1m4s
5	Simple cycle (9x)	108	0.4s
6	Simple method call (3x)	50	0.3s
7	Simple cycle (3x)	9	0.2s
8	Simple method call (2x)	4	0.2s
9	Simple cycle (2x)	1	0.2s
10	Redundant state (17x)	1	0.2s

V. Case Study

In this section, we will share with the reader our experience with applying the reduction rules presented in Sect.III to a nontrivial demo application developed in one of our projects [13] (Fig. 1).

As a proof of the concept, we have taken one of the tests specifications prepared in the project (Fig. 15). The test consists of a consent composition of the following components: $Arbitrator, Token, AccountDatabase, CardCenter$, and $Firewall$. Without applying the rules, the verification required to visit 871122 states and it took about 2 hours and 23 minutes in total to traverse them; see Tab. I. The table also illustrates the effect of applying the reduction rules.

First, external events (26 alltogheter) were to be replaced by ν. Ten of them could be removed immediatelly, such as $?IArbitratorLifetimeController.Start\uparrow$ and $?Ilogin.GetTokenIdFromIpAddress\uparrow$. However, removal of the rest of them was prevented by conditions of ν-elimination (Sect.III-A); in particular by the fact, that no accept transition can begin from the state which is the end of a ν-transition. Fortunately, these "preventing" transitions were also external events which could be removed. After all of these reductions, the verification took less than 5 minutes and required approximately 36000 states to traverse.

In the following step (2), ten simple method calls were reduced, for example $IFirewall.DisablePortBlock$ and $ICardCenter.Withdraw$. After that, in the step (3) initial state transitions $!ITokenLifetimeController.Start\uparrow$ and $?ITokenLifetimeController.Start\uparrow$ were eliminated. The status of the specification after this step finished is in Fig. 16.

Further, the topmost redundant states in the components $Arbitrator$, $Token$, $AccoutDatabase$ and $Firewall$ were reduced by applying the rule in (Fig. 14c). After this reduction, the number of the topmost LTSs rose from five to 17. In the next step (5), nine simple cycles such as $?IcardCenter.Withdraw\uparrow*$ were reduced. At the first sight surprisingly, after applying additional reduction rules (steps 6-10 in Tab. I) only a single state for each LTS remains, and is finally removed as redundant.

Arbitrator

```
( ?IArbitratorLifetimeController.Start^ ;
  !ITokenLifetimeController.Start^ ;
  [?ITokenLifetimeController.Start$,
    !IArbitratorLifetimeController.Start$] );
(
  (
    ?ILogin.GetTokenIdFromIpAddress +
    ?ILogin.LoginWithFlyTicketId {
      !IFlyTicketAuth.CreateToken ;
      (!IFirewall.DisablePortBlock + NULL) }
    +
    ?ILogin.LoginWithFrequentFlyerId {
      !IFreqFlyerAuth.CreateToken ;
      (!IFirewall.DisablePortBlock + NULL) }
    +
    ?ILogin.LoginWithAccountId {
      !IAccountAuth.CreateToken ;
      (!IFirewall.DisablePortBlock + NULL) }
    +
    ?ILogin.Logout {
      !IToken.InvalidateAndSave_1 }
  )* |
  ?ITokenCallback.TokenInvalidated_1 {
    !IFirewall.EnablePortBlock_1 }*
  |
  ?ITokenCallback.TokenInvalidated_2 {
    !IFirewall.EnablePortBlock_2 }*
  |
  ?ITokenCallback.TokenInvalidated_3 {
    !IFirewall.EnablePortBlock_3 }*
  |
  ?IDhcpCallback.IpAddressInvalidated_1 {
    !IToken.InvalidateAndSave_2 }*
)
```

Token

```
?ITokenLifetimeController.Start ;
( ?IToken.InvalidateAndSave_1 {
    (!IAccount.AjustAccountPrepaidTime_1 + NULL);
    !ITokenCallback.TokenInvalidated_1 }* |
  ?IToken.InvalidateAndSave_2 {
    (!IAccount.AjustAccountPrepaidTime_2 + NULL);
    !ITokenCallback.TokenInvalidated_2 }* |
  (
    (!IAccount.AjustAccountPrepaidTime_3 + NULL);
    !ITokenCallback.TokenInvalidated_3 )*
)
```

AccountDatabase

```
( (
    ?IAccount.GenerateRandomAccountId +
    ?IAccount.CreateAccount +
    ?IAccount.RechargeAccount {
      !ICardCenter.Withdraw }
  )* |
  ?IAccount.AjustAccountPrepaidTime_1* |
  ?IAccount.AjustAccountPrepaidTime_2* |
  ?IAccount.AjustAccountPrepaidTime_3*
)
```

CardCenter

```
( ?ICardCenter.Withdraw* )
```

Firewall

```
( ?IFirewall.EnablePortBlock_1* |
  ?IFirewall.EnablePortBlock_2* |
  ?IFirewall.EnablePortBlock_3* |
  ?IFirewall.DisablePortBlock*
)
```

Fig. 15. System specification: arbitrator.bp

Arbitrator

```
(
  (
    (!IFirewall.DisablePortBlock^ + NULL)
    +
    (!IFirewall.DisablePortBlock^ + NULL)
    +
    (!IFirewall.DisablePortBlock^ + NULL)
    +
    !IToken.InvalidateAndSave_1
  )*
  |
  ?ITokenCallback.TokenInvalidated_1 {
    !IFirewall.EnablePortBlock_1^ }*
  |
  ?ITokenCallback.TokenInvalidated_2 {
    !IFirewall.EnablePortBlock_2^ }*
  |
  ?ITokenCallback.TokenInvalidated_3 {
    !IFirewall.EnablePortBlock_3^ }*
  |
  !IToken.InvalidateAndSave_2*
)
```

Token

```
(
  ?IToken.InvalidateAndSave_1 {
    (!IAccount.AjustAccountPrepaidTime_1^ + NULL);
    !ITokenCallback.TokenInvalidated_1 }*
  |
  ?IToken.InvalidateAndSave_2 {
    (!IAccount.AjustAccountPrepaidTime_2^ + NULL);
    !ITokenCallback.TokenInvalidated_2 }*
  |
  (
    (!IAccount.AjustAccountPrepaidTime_3^ + NULL);
    !ITokenCallback.TokenInvalidated_3 )*
)
```

AccountDatabase

```
( !ICardCenter.Withdraw^* |
  ?IAccount.AjustAccountPrepaidTime_1^* |
  ?IAccount.AjustAccountPrepaidTime_2^* |
  ?IAccount.AjustAccountPrepaidTime_3^*
)
```

CardCenter

```
( ?ICardCenter.Withdraw^* )
```

Firewall

```
( ?IFirewall.EnablePortBlock_1^* |
  ?IFirewall.EnablePortBlock_2^* |
  ?IFirewall.EnablePortBlock_3^* |
  ?IFirewall.DisablePortBlock^*
)
```

Fig. 16. Situation after step 4

VI. Discussion and Related Work

Even though the results of the case study are persuasive, two obvious questions have to be answered to claim a real benefit of the presented reduction process.

a) In which order are the rules to be applied (and how many times): Based on experiments, the rule of thumb is that the Rules B4 and C1 reflect are to be applied first and only once (in any order). This is because they are driven by the static component architecture and do not consider actual relationship among method calls.

On the other hand, all the remaining rules depend on each other. For example, applying the rule B1 on two consecutive method calls will enable the rule B2 or B3 to be applied. At the same time, applying the rule B2 may enable B3 while applying the rule B3 may enable B1 and B4.

In general, it is not easy to specify a correct order of reductions to achieve minimal state space. However, for a class of specifications created by method abbreviations only, the best result is achieved by the same ordering of reduction rules as chosen in Sect. III. Fortunately, behavioral constructs outside the scope of method calls or acceptances are rare.

b) Is the result of reduction always a composition of empty protocols, similar to the case study: It is relatively easy to show that the answer is no. Consider for instance the composition of the protocols: $(?c\uparrow; !d\uparrow)*; ?a\uparrow \triangledown (?d\uparrow; !c\uparrow)*; !a\uparrow$ — there is no way to reduce them by static analysis. However, our experiments indicate that in all real case studies we had available, there is always a substantial reduction in the state space size, in particular when the B1 rule is repeatedly applied. On our future work list, there is the search for frame protocol classes which guarantee emptiness of the reduction result.

Related to our work are slicing and symmetry exploiting methods. In *program slicing*, we define *slicing criterion* which is typically a pair of program location and a set of variables. Then, a program slice is a part of the original program which affects the values of presented variables at the specified location.

The idea of program slicing was introduced by Weiser [14], [15] originally for debugging purposes. Later, program slicing has been used for various purposes such as analysis, parallelization, comparison, and testing. With respect to our work, the most interesting topic is compiler optimization. Larus and Chandra [16] present a detection of redundant common sub-expressions. The code is enriched by instructions for trace inspection. Traces are then interpreted as a stream of events, where a directed acyclic graph is constructed with all the arguments and operators which affects the current value in the register. Because the approach uses information about executions, it represents a class of *dynamic* slicing.

Specification slicing aims at creating a reduced specification while preserving all the desired information. It is analogous to *program slicing* for specifications. Wu and Yi [17] present slicing of Z [18] specifications. First, data, control, and logic dependencies of the specification are represented by a specification dependence graph. Then, the reduced Z specification is created by a two-pass reachability algorithm applied to the graph.

Another work (done in our group) [19] presents slicing of the component behavior specification according to the actual component composition. The technique aims at removing the unused behavior to make the actual real role of a particular component more visible by removing the unemployed parts of the frame protocols. Thus, although the size of the specification is reduced, the real size of the state space is untouched, since only its employed part contributes to the size of the consent composition.

A key difference between slicing and our reduction method is that slicing reduces the unemployed part of the specification, while our method is specialized towards consent composition and reduces also the parts of the specification for which composition without communication errors can be statically guaranteed. At the same time, the method guaranties that, by the reduction, no communication errors will be injected and no communication error will be eliminated from the specification. Moreover, our reduction method involves also logging of the partial reduction steps to ensure the original form of the specification can be reconstructed - this is necessary for providing counter examples referring to the original form if the reduction does not result in an empty specification and model checking has to be applied in its rest. This is not strictly required in a case of slicing.

There are several approaches to exploit symmetry in the specification. For example in Murϕ model checker [20], Ip a Dill introduced [21] a new data type *scalarset* which represents and unordered set. Operations on scalarset are restricted to guarantee that every function on the set is automorphism. Thus, scalarset is symmetric and the behavior of the program is independent on the actual permutation of elements. Additionally, conditions under which the scalarset can be used are statically checked.

VII. Conclusion

We attack the state space explosion problem by reducing the specification to be verified. The reduction is done by an iterative application of reduction rules, which have been articulated with respect to the consent composition. To guarantee that the reductions do not inject additional compositional errors and do not eradicate a compositional error present in the specification, we have formulated several conditions to be satisfied. Reductions are of a low overhead, since the algorithm is linear with the size of the specification.

Although a ”full” reduction of the specification is not guaranteed in general, the real-life case studies, such as the one presented in this paper, show that most of such specifications contain several typical patterns to which reduction rules apply. If the result of the reductions is not an empty protocol and, therefore, a standard model checking verification is to be applied on the rest, the potential counter example is back interpreted in the original specification. This makes the reduction method fully transparent to the user.

ACKNOWLEDGMENT

This work was partially supported by the Czech Academy of Sciences project 1ET400300504.

REFERENCES

[1] F. Plasil, S. Visnovsky, and M. Besta, "Bounding component behavior via protocols," in *TOOLS*, vol. 30. USA: IEEE Computer Society, Aug 1999, pp. 387–398.

[2] J. Adamek and F. Plasil, "Component composition errors and update atomicity: Static analysis," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 17(5), pp. 363–377, Sep 2005.

[3] J. A. Bergstra and J. W. Klop, "Process algebra for synchronous communication," *Information and Control*, vol. 60, no. 1-3, pp. 109–137, 1984.

[4] R. Milner, *A Calculus of Communicating Systems*, ser. Lecture Notes in Computer Science. Springer, 1980, vol. 92.

[5] M. Mach, F. Plasil, and J. Kofron, "Behavior protocol verification: Fighting state explosion," in *International Journal of Computer and Information Science*, vol. 6. ACIS, Mar 2005, pp. 22–30.

[6] P. Parizek, F. Plasil, and J. Kofron, "Model checking of software components: Combining Java PathFinder and behavior protocol model checker," Charles University, Tech. Rep. 2006/2, Jan 2006.

[7] V. Holub and P. Tuma, "Streaming state space: A method of distributed model verification," in *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*. IEEE Computer Society, 2007, pp. 356–368.

[8] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Los Angeles, California: ACM Press, 1977, pp. 238–252.

[9] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 5, pp. 1512–1542, Sep 1994.

[10] D. Peled, "Combining partial order reductions with on-the-fly model-checking," in *Proceedings of the 6^{th} International Conference on Computer Aided Verification*, ser. Lecture Notes In Computer Science, D. L. Dill, Ed., vol. 818. Springer-Verlag, Jun 1994, pp. 377–390.

[11] A. Valmari, "A stubborn attack on state explosion," in *Proceedings of the 2^{nd} Workshop on Computer Aided Verification (CAV'90)*, ser. Lecture Notes in Computer Science, E. M. Clarke and R. P. Kurshan, Eds., vol. 531. London, UK: Springer-Verlag, Jun 1991, pp. 156–165.

[12] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2001.

[13] P. Jezek, J. Kofron, and F. Plasil, "Model checking of component behavior specification: A real life experience," in *International Workshop on Formal Aspects of Component Software (FACS'05)*, vol. 160. Elsevier B.V, Aug 2006, pp. 197–210.

[14] M. Weiser, "Program slices: formal, psychological, and pratical investigations of an automatic program abstraction method," Ph.D. dissertation, University of Michigan, 1979, ann Arbor.

[15] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, vol. 10, no. 4, pp. 352–357, 1984.

[16] J. R. Larus and S. Chandra, "Using tracing and dynamic slicing to tune compilers," University of Wisconsin-Madison, Tech. Rep. CS-TR-1993-1174, 1993.

[17] F. Wu and T. Yi, "Slicing Z specifications," *SIGPLAN Notices*, vol. 39, no. 8, pp. 39–48, 2004.

[18] M. Spivey, *Z Notation*. Prentice Hall, Jun 1992.

[19] O. Sery and F. Plasil, "Slicing of components' behavior specification with respect to their composition," in *10th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE 2007)*, Jul 2007.

[20] D. L. Dill, "The Murφ verification system," in *CAV '96: Proceedings of the 8^{th} International Conference on Computer Aided Verification*. London, UK: Springer-Verlag, 1996, pp. 390–393.

[21] C. N. Ip and D. L. Dill, "Better verification through symmetry," in *CHDL '93: Proceedings of the 11th IFIP WG10.2 International Conference*. Amsterdam, The Netherlands, The Netherlands: North-Holland Publishing Co., 1993, pp. 97–111.

Chapter 7

On Distributed Verification of Generalized Interaction Models of Software Components

Viliam Holub

Presented paper at **20th European Conference on Object-Oriented Programming (ECOOP 2006) Doctoral Symposium** [5].

On Line: http://www.ecoop.org/phdoos/ecoop2006ds/ds/holub.pdf

On Distributed Verification of Generalized Interaction Models of Software Components*

Viliam Holub

Distributed Systems Research Group
Faculty of Mathematics and Physics, Charles University in Prague

Abstract. The presented work aims at developing a distributed verification tool for a generalized automata system. The verification is done by an exhaustive state-space exploration – a stream of linearly-organized system states is updated on each node of a computational cluster and handed over to the next node over a network. The automata system is proposed to be general enough to describe common automata-based formalisms used for describing interaction of software components. We also present results obtained with the proof-of-concept implementation.

1 Motivation and Problem Statement

Model checking of software components is an approach to automatically verify selected aspects of component correctness. As software itself is very difficult to verify, abstracted models and constraint languages have been introduced. We focus on those models and languages, that are based on automata and interactions among them, such as the specification of the transition rules and component composition. In behavior protocols[1] these models and languages allow us to check rules and bounds on interface utilization, and even to check complex interactions in the entire component system based on a complete description of possible actions and reactions.

The existing approaches to model-checking of complex software architectures are trying to overcome two major problems. One is the problem inherent to the complexity of the model – the number of states of a model tends to grow exponentially with the size of the model, leading to a situation where the model either does not fit into the available memory, or the time to visit the states of the model is too long. This is referred to as a state explosion. The other is the problem inherent to the complexity of the model-checking methods – the methods consist of complex algorithms that are often implemented as a proof of concept only, thus lacking the potential an optimized implementation would bring.

One of the approaches to solve the state space explosion problem is distribution of the model checking algorithm. The typical method is dividing the state space into distinct parts[2], where each node checks only the assigned states and, if necessary, asks the other nodes to visit other reachable states as the model

* This work was supported by the Czech Science Foundation (GACR) 201/05/H014.

checking algorithm traverses the state space. The network latency and load becomes a bottleneck, giving rise to a hard problem of optimal state distribution. Moreover, due to random access to states, the state space cannot be effectively saved on mass storage devices.

Our goal is to solve these problems, namely:

- To develop an effective distributed verification tool based on exhaustive exploration of linearly-organized state space and error state reporting. We restrict the problem to systems that describe interactions among components, are based on Label Transition Systems (LTS), and have a finite number of states. Because of limitations on the model, we can use more powerful optimizations than similar systems based on complex formalisms (LTL, CTL, models with unlimited state space).
- To design a formal model of component interaction should the existing models prove unsuitable. The model should be general enough to allow describing a majority of common constructs available on LTS and be suitable for effective automated verification.
- To present the approaches to conversion from other formal models to the verified one and to show how to interpret the results.

2 Related Work and Model Analysis

The related work discusses ADL-based approaches to modeling, the abstract modeling approaches based on automata, and the problem of model verification.

The Wright Architectural Description Language[3] uses a process algebra (a subset of CSP) for defining component behavior. The tested property is called compatibility[4] and in short says that the connector interaction cannot detect that the role process has been replaced by the port process. Behavior protocols[1] (BP), embedded in SOFA[5], allow the simplification of the verification process by hierarchical decomposition of the architecture, starting with the smallest parts of the system and continuing in the bottom-up manner, thus reducing the internal complexity of composite components. BP allows identifying several composition errors, including no activity (deadlock) and bad activity. Furthermore, it defines compliance[6] as the relation of expected and real behavior.

Interface automata[7] allow compatibility checks between interface models in optimistic view, i.e. components can be used together if there is at least one correct design. The composition is defined on two automata only, synchronizing on one input and one output action. Team automata[8] is a composition of component automata. The specified action is executed in the team by simultaneously executing the action in all component automata. There is no limitation for the number of states. Component-interaction automata[9] has been developed with respect to ADL and software components. The concept is similar to team automata; however there are differences the authors give reasons to be more suitable for software design.

Although a large spectrum of formalisms has been mentioned and analyzed, none of them is both general enough and easy to verify. Thus, we have decided

to propose a new model denoted later in the text as *interference automata*. It is close to team automata.

Notably, interference automata are not limited to one type of synchronization (synchronous, asynchronous or blocking, non-blocking) nor a number of action participants (one-to-one, one-to-many, and many-to-many). Having this in mind, the model can be outlined as a set of local automata, augmented by a set of transition rules and a set of alarms. A system state is a vector of states of all the local automata. A transition is a pair of starting and final state. For practical reasons, a range of starting and final states is allowed as well. Alarms describes states which are somehow "interesting" and its reachability should be reported (often it is related to erroneous states). The proposed model does not cover all the needs of the mentioned formal methods (for example, the unlimited number of states), but fits perfectly to our requirements.

There are many verification tools (including SPIN[10] and DiVinE[11]) that aim mainly at checking LTL or CTL formula. SPIN uses the Depth First Search (DFS) to explore the state space and uses the operational memory as a state cache to prevent a re-exploration of previously visited states. Because a random access to data structures is required, external slow devices such as hard-disks cannot be used effectively. We believe that targeting simpler interaction protocols and relations among components allows for a more efficient verifier implementation.

3 Distributed Verification

We face the need of storing a large amount of states via a special state space encoding. All properties (such as *state-explored* and *state-to-be-explored*) of reached states are stored sequentially in the order of the DFS traversal algorithm, creating a stream. Because a numeric value of the state can be computed from its position in the stream, it is omitted. This approach saves a significant amount of storage space. The stream can be passed across computational nodes organized in a logical circle, allowing all the nodes to work simultaneously on different parts of the stream and systematically updating the state space. Because of the linear access pattern, buffering on mass storage devices is possible with nearly zero-overhead.

A drawback of this approach is the need to avoid random access to the stream. During the verification process, nodes access the states only sequentially and use their operational memory as a state cache and a state buffer. In the case of memory overflow, the node has to suspend the exploration, store the explored states from the state buffer, and thus free the sufficient amount of memory.

Because forced flushing of the state buffer reduces verification effectivity, three basic techniques to reduce memory consumption are used. The first technique is a static optimization which affects each component of the system individually (automata minimalization). A dynamic optimization takes into account the interactions among automata, trying to identify unreachable system states and remove them from the model. This improves the state space compression.

The third optimization is applied on-the-fly during the verification process. A coherent part of the explored state space with the same properties is replaced by a single cut-off superstate.

We plan to prove the concept on a real-life project[12]. At present, a basic, single-node prototype without dynamic optimizations shows promising results. For instance, the average amount of explored states is about 5 millions states per second (sps) on a low-end desktop, compared to 180000 sps of the SPIN model checker. This will change as more optimizations will be included. On a testing example with 10^7 states, the overall memory requirements were 6 bits per state.

Future work will focus on implementation of a parallel reduction technique (with less overhead than classical LTL and CTL verifiers have) and further static model optimizations.

References

1. Plasil, F., Visnovsky, S., Besta, M.: Bounding component behavior via protocols. In: TOOLS. Volume 30., USA, IEEE Computer Society (1999) 387–398 1, 2
2. Garavel, H., Mateescu, R., Smarandache, I.: Parallel state space construction for model-checking. In: SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software, New York, NY, USA, Springer-Verlag (2001) 217–234 1
3. Allen, R.J.: A Formal Approach to Software Architecture. PhD thesis, Carnegie Mellon University (1997) Chair-David Garlan. 2
4. Allen, R., Garlan, D.: A formal basis for architectural connection. ACM Transactions on Software Engineering and Methodology (TOSEM) **6** (1997) 213–249 2
5. Plasil, F., Balek, D., Janecek, R.: SOFA/DCUP: Architecture for component trading and dynamic updating. In: International Conference on Configurable Distributed Systems (CDS), Washington, DC, USA, IEEE Computer Society (1998) 2
6. Adamek, J., Plasil, F.: Erroneous architecture is a relative concept. In: Software Engineering and Applications (SEA) conference, Cambridge, MA, USA, ACTA Press (2004) 715–720 2
7. de Alfaro, L., Henzinger, T.A.: Interface automata. In: Ninth Annual Symposium on Foundations of Software Engineering (FSE), New York, NY, USA, ACM Press (2001) 109–120 2
8. Ellis, C.: Team automata for groupware systems. In Hayne, S., Prinz, W., eds.: SIGGROUP Conference on Supporting Group Work: The Integration Challenge (GROUP), ACM Press, New York (1997) 415–424 2
9. Zimmerova, B., Brim, L., Cerna, I., Varekova, P.: Component-interaction automata as a verification-oriented component-based system specification. SIGSOFT Software Engineering Notes **31** (2006) 2
10. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Software Engineering (TSE) **23** (1997) 279–295 3
11. Barnat, J., Brim, L., Cerna, I., Simecek, P.: DiVinE the distributed verification environment. In Leucker, M., van de Pol, J., eds.: 4th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC), Lisbon, Portuga (2005) 3

12. Jezek, P., Kofron, J., Plasil, F.: Model checking of component behavior specification: A real life experience. In: International Workshop on Formal Aspects of Component Software (FACS'05). (2005) 4

Chapter 8

Streaming State Space: A Method of Distributed Model Verification

Viliam Holub and Petr Tůma

Contributed paper in **The First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering** [2].

In conference proceedings,
published by IEEE Computer Society,
pages 356–368,
ISBN 978-0-7695-2856-4,
2007.

The original version is also available electronically from the publisher's site at http://dx.doi.org/10.1109/TASE.2007.47.

Streaming State Space: A Method of Distributed Model Verification (pre-print)

Viliam Holub and Petr Tuma
Distributed Systems Research Group
Faculty of Mathematics and Physics
Charles University in Prague, Czech Republic
{holub,tuma}@dsrg.mff.cuni.cz

Abstract

We present an alternative to traditional approaches of parallel and distributed model verification. In contrast to methods based on partitioning the state space, we keep the states together, sorted and organized as a stream. The stream is passed across computational nodes in a logical circle where each node updates the states in the part of the stream it currently sees. The presented method relies on state locality, allows storing the state space at external storage devices, and scales well on a non-dedicated, non-uniform cluster of up to tens of nodes. We have implemented an experimental version of the algorithm and obtained very promising results in scalability in the number of processors and computational nodes used in a large real-life use case.

1 Introduction

Model verifiers based on an explicit state space enumeration have to face the problem of large state space size [18]. The size of the state space tends to grow exponentially with the size of the model specification (the *state explosion problem*) and is one of the main obstacles in practical use of model verification for real-life systems. Together with other methods aimed at reducing the number of states that have to be generated [22] or stored [6], the time consumption can be significantly reduced by modifying the state space organization and the verification algorithm so that all the available computational resources are employed optimally.

Contemporary computers have several independent execution units (a combination of multiprocessor, hyperthreading or multicore technologies), are equipped with network cards and hard drives. In a verification tool based on exhaustive state space enumeration, we would like to take advantage of modern equipment. Particularly, we would like to achieve:

- *parallelization* – concurrent computation on more than one processor of a computational node (also referred to as tightly coupled processors, most often symmetric multiprocessors, SMP). Parallel computing becomes a significant direction in future hardware development, and, partly due to power consumption, the number of available processors is likely to increase more than the raw performance of each single processor.

- *distributability* – computational nodes are connected via network (referred to as loosely coupled processors). This type of computational power aggregation is characterized by expensive access to memory of other nodes. In contrast with parallelism, increasing the computational power by adding nodes is even cheaper than by adding processors.

- *use of external storage devices* – these are significantly cheaper than main memory and provide larger storage space. The main drawback here is a relatively slow access to the device, and even worse, the random access latency is very high [4].

We are interested in computing clusters made of common desktop computers. The computers do not have to be dedicated for verification use, we would like to utilize the idle time during a common user workload. Such environment is very non-uniform in the terms of available performance, memory size and disk space. In a medium size organization, we expect tents of computers connected via 100Mbs or 1000Mbs Ethernet to be available.

A traditional approach to distributed model verification (Sect. 2.2) is to partition the state space into regions according to the number of computers (computational nodes) in the cluster. The partitioning function is designed to minimize the necessary network communication. We have chosen a different approach. In this paper, we present a method of circulating the whole state space across nodes in a logical circle (Sect. 3.1), inspired by the fact that a streamed workload fits the modern computers perfectly. Because such

an approach raises questions about effectivity, our aim was also to experimentally measure the real performance of the method on very large theoretical and practical examples.

First, we outline the necessary background (Sect. 2). This includes an overview of the exploration process, the traditional approaches to distributed model verification and their drawbacks. After that, we explain the ideas behind our method (Sect. 3) and describe our prototype implementation. We then present (Sect. 4) and evaluate (Sect. 5) experience from our case studies. Finally, we discuss the related work (Sect. 6) and summarize the contribution (Sect. 7).

2 Background

2.1 General exploration algorithm

The *state space exploration* (or simply *exploration*) is a process of systematic enumeration of all the system states. Its goal is to decide which states are reachable in the formal model and report those reachable states that have certain sought-for properties. The properties of a state that we are searching for vary among different formal models and include a failure to satisfy a condition or an existence of a deadlock.

The traversal algorithm is sketched in Alg. 1. The exploration process begins in the starting state (step 1) and proceeds by the repeat loop (the *exploration step*, steps 2-9). We use two permanent state sets, V and E, and one temporary state set S. The set V contains states that are reachable from the starting state, but whose successors have not been enumerated yet. These states are referred to as *open*. After all successors of a state have been enumerated (step 5), the state becomes *closed* and is moved from V to E (steps 6, 7, and 8). The algorithm stops when all open states have been closed, that is when $V = \emptyset$ (step 9).

Algorithm 1 General traversal algorithm.

1: Set the initial values
   $E = \emptyset$, $V = \{\text{starting state}\}$, $S = \emptyset$
2: **repeat**
3: Take one state s from V
4: Report s if it has the sought-for properties
5: Generate all successors of s and store them in S
6: Add s to closed states E
7: Remove already closed states ($\in E$) from S
8: Add S to open states V and empty S
9: **until** $V \neq \emptyset$

Specific instances of the general traversal algorithm differ mostly in the data structures used and in the approach to choosing open states for the exploration (step 3). Among the most common are Depth First Search [12] (DFS) and Breadth First Search [19] (BFS).

DFS keeps the active path in a stack and the states selected in step 3 are successors of the state on the top of the stack. DFS is relatively easy to implement and is used primarily in non-distributed model checkers [12]. The main disadvantages of the algorithm are long counter-examples produced [21] and a relatively complicated implementation in a distributed environment [24, 3].

In BFS, states in step 3 are selected based on their distance from the starting state. The algorithm is preferred in distributed model checking, partly because of easier parallelization. The main advantage of BFS is that the counter-example returned is the shortest possible. Unfortunately, strict BFS requires synchronization, which may reduce the overall performance [13]. That is why some researchers propose algorithms that only follow BFS in principle, but break the strict BFS ordering in situations where the exact behavior would cause performance penalty.

2.2 Traditional approach to distributed verification

A widely adopted method of distributed verification is to partition the state space into several disjunct parts corresponding to the number of available computational nodes [25] and assign each part to one node. A node visits only the assigned states and, if it reaches a state outside the assigned part, it asks the owner of the corresponding part to visit the state. In other words, for every new state s a function `partition(s)` is evaluated. If the result belongs to the owned state part, it is processed locally. Otherwise, the request for visiting the state is sent in a message to the corresponding state space part owner.

The choice of the partition function is crucial for an efficient distribution. The best partitioning function (as discussed in [17]) should divide the space evenly and minimize the number of cross-region transitions. A badly chosen function may lead to workload asymmetry, increased communication overhead, and thus induce a performance penalty. Because the state space is typically not homogeneous, mostly due to the structure of the associated model specification, finding the optimal state space division is a problem as hard as the verification itself (see Sect. 6 for further details).

The problem of dividing the load still persists in non-uniform clusters and clusters of workstations. Fast computers will wait for their slower counterparts or, if assigned a larger portion of the state space, may reach their memory limits. Especially in clusters of workstations, performance of individual computers continuously varies due to user workloads, which cannot be compensated for with a static partitioning function. Thus, most approaches target dedicated uniform clusters, while the specific properties of non-uniform clusters are rarely addressed [15].

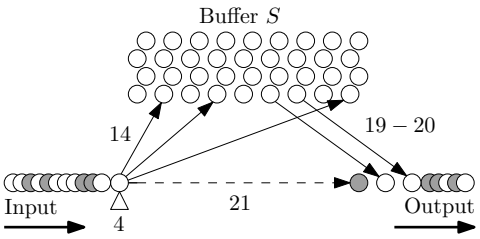


Figure 1. Streaming state space: data flow

3 State space streaming

3.1 Concept

Instead of partitioning the state space (Sect. 2.2), we circulate the whole state space across all nodes in a logical circle. States are kept sorted and passed sequentially, forming a stream (hence the name *streaming*). Each node receives the stream of states from its predecessor, updates it, and hands the updated stream over to its successor. A part of the stream can be temporarily held in the node memory or buffered on an external storage device. One passage of the stream through the logical circle is referred to as an *epoch*.

During the computation (the algorithm is sketched in Alg. 2 and illustrated with a dataflow diagram in Fig. 1), we keep a buffer of generated states S, the last state passed to the output $last$, and the epoch number $epoch$. In the general exploring algorithm (Sect. 2.1), the open and closed states were kept in separate sets. Here, we keep the open and closed states together, distinguished by an additional $status$ flag assigned with each state.

The first node in the logical circle (selected arbitrarily) starts the exploration by forming an initial stream consisting of the starting state and the end-of-stream tag. After the initialization (step 1), the algorithm starts reading states from the input stream (4). If the read state s is open (5), we generate its successors (6), remove duplicates of s (7), and, marked as open, we store the successors in the buffer S (12-15). If the state s has the sought-for properties, we mark it as `sought-for` and inform the user (8-10). Otherwise, we mark it as closed. Before we store s to $last$ and pass the state s to the output stream (21-22), we extract all the states within the interval $last..s$ from the buffer S and write them, sorted, to the output stream. As we reach the end-of-stream tag (3), we pass it to the output, set $last$ as a starting state and increase the $epoch$ index (24-26).

Our method follows BFS in principle, but not strictly. During one epoch, when all the successors of open states are generated, a subset of them may be included into the same epoch. That is due to the independence of the exploring (steps 6-7 and 12-15) and updating (steps 19-20) operations.

Algorithm 2 Streaming state space

```
1:  Create a stream with the starting state followed by the
    end-of-stream tag (first node only)
    S = ∅
    last = starting state
    epoch = 0
2:  loop
3:    while not end-of-stream on input do
4:      Take a state s from the input
5:      if s is open then
6:        Generate all successors of s into Z
7:        Remove s from S and Z
8:        if s has the sought-for properties then
9:          Mark s as sought-for and successor of s
10:         Report s
11:       else
12:         for all z : z ∈ Z ∧ z ∉ S do
13:           Mark z as open and successor of s
14:           Add z to S
15:         end for
16:         Mark s as closed
17:       end if
18:     end if
19:     Extract X = {∀x ∈ S : last < x ≤ s} \ {s}
20:     Put ordered and sorted X on output
21:     Put s on output
22:     Set last = s
23:   end while
24:   Pass end-of-stream tag from input to output
25:   Set last = starting state
26:   Increase epoch
27: end loop
```

3.2 Termination

The exploration finishes when no open state exists in the state stream or in the node buffers. Unfortunately, this is not an easily testable condition in a distributed environment, where a consistent snapshot of the state stream and the node buffers cannot be obtained efficiently due to message latency and synchronization constraints. A solution used in our experimental implementation is to keep a flag associated with the end-of-stream tag (here simply a tag). The flag is cleared by the first node in the logical circle, which also counts the number of open states passed to its successor in the logical circle during an epoch. When a node receives the tag and its buffer is not empty, it sets the flag, otherwise it passes the flag on unchanged. When the first node receives the tag with the flag cleared and the number of open states in the epoch is zero, the exploration is complete and all participants are directed to stop passing the stream.

3.3 Counter-example extraction

Apart from its status, each state is associated with information about its predecessor (steps 9 and 13). Thus the counter-example path can be extracted from the stream beginning at the sought-for state and continuing backwards to the starting state. One predecessor per state is enough for successful reconstruction of one counter-example. Moreover, because we always keep the first predecessor of the state, the counter-example path is likely to be the shortest one.

In our implementation, after stopping the circulation of the stream, we query all nodes to obtain the range of states that they hold. We then ask the owner of the sought-for state for the information about its predecessor, and repeat this step recursively until the initial state is reached. Because the stream is sorted, a state can be located quickly within a node using binary division search.

Alternatively, the predecessor states can be stored on a hard drive [25] in order to reduce the stream size.

3.4 Sorting

Except for transitivity and reflexivity, the algorithm imposes no restrictions on the ordering function used for the sorting of states (steps 19-20). Preferably, the function should take locality into account – for example by using a basic lexicographic ordering and prefixing the states with a value of a traditional partitioning function from Sect. 2.2.

Additionally, as we have observed during our experiments, even a simple lexicographic ordering of states speeds up an otherwise unoptimized generation of successors. This is an effect of real hardware behavior, memory caches in particular.

3.5 Buffer organization

Because we need to extract a range of states from the buffer S frequently (step 19), it is beneficial to keep the buffer in a specialized data structure. One example is maintaining two separate heaps, with states that are greater (resp. smaller) than $last$ added to the first (resp. second) heap in step 14. In steps 19-20, we can then easily extract all the states smaller than s from the top of the first heap and write them directly to the stream. Heaps are exchanged at the end of the stream. Accesses to the states have logarithmic complexity.

To handle very large state spaces, only a part of the buffer (states close to s) has to be accessible in node memory. The rest of the states can be stored on an external storage device, freeing the node memory.

3.6 External storage

The stream grows during the verification process and can easily overflow the available memory. We can offset the overflow limit by storing the stream temporarily on an external storage device. As we get the states from the predecessor, we immediately initiate an asynchronous write to the external storage device. The states are then retrieved as necessary by the verification algorithm.

Since we access the stream linearly and thus predictably, we can mitigate the impact of high access latency. The states are manipulated in large chunks to reduce the number of seeks and are read in advance so that the verification proceeds smoothly, without access-latency-induced delays.

3.7 Load balancing

In non-uniform clusters and especially in clusters of workstations, the performance of a node may vary owing to the hardware configuration and the actual user load. Optimally, the process of verification would load all the computation nodes to their maximum capacity. In our case, the key requirement behind this goal is that the algorithm should not block because of shortage of input states. In other words, if a node is about to block, its predecessor in the logical circle has to speed up.

To meet this requirement, we need to adjust the speed of the stream passage through a node. The exploration algorithm has two time-expensive parts – the first one generates successors and adds them into a buffer (steps 5-18), the second one extracts the states from the buffer and writes them to the output stream (19-20). A node can skip both parts at will, slowing down the progress through the state space but preserving the correctness of the exploration. Because the state extraction part adds open states to the stream and thus increases the workload for the successor node, it is beneficial to reduce the workload smoothly by skipping the exploration of open states with certain probability.

The workload balancing in our implementation works as follows. Every node keeps a parameter p in the range [0..1], initialized to 0. Each node skips the exploration part of a state with the probability of p. When a node encounters a shortage of input states, it sends a *hurry up* message to its predecessor. A node that receives the message increases the probability of skipping states by 50%: $p = p+(1-p)/2$. To address dynamic changes in the environment, p is decreased by 0.1 towards zero periodically: $p = \max(0, p - 0.1)$.

Additionaly, when states are missing on the input, a node may start to explore open states stored in the buffer: in step 4, s would be removed from the buffer, after step 18, s would be returned back to the buffer, and the algorithm would continue with step 3. The duplicate state detection is thus postponed.

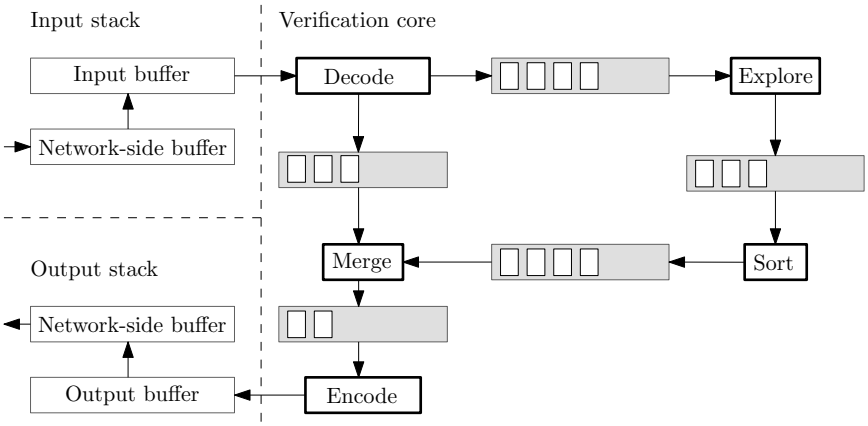


Figure 2. The internal structure of a computational node.

3.8 Implementation details

In this section, we share the details of our experimental proof-of-concept implementation with the reader. In the implementation, we verify models specified by the Interfering Automata [11], suitable for modelling Behavior Protocols [1, 2, 23]. The internal structure of the implementation is displayed on Fig. 2 and corresponds to the data-flow diagram on Fig. 1.

Our idea is to spawn the same number of execution threads (workers) as is the number of available processors and assign the work dynamically. We have divided the algorithm into five distinct operations – *decode*, *explore*, *sort*, *merge*, and *encode*.

The *decode* operation decodes the states from the network and stores them in a host-friendly format. This is required because of different hardware architectures among the nodes in a cluster. The resulting stream of decoded states (still sorted) is forwarded to the input of the merge operation. Open states are marked as closed (corresponds to step 16 in Alg. 2) and additionally copied to the input of the explore operation.

The *explore* operation generates successors of the input states and accumulates them in the output buffer (step 6). It also tests the states for the criteria being verified on the model; these states are reported to the user (steps 8-10).

For the purposes of duplicate state detection and the addition of the generated states to the stream (steps 19-20), the *merge* operation forms the output stream by combining the original input stream and the newly generated states in the buffer. Because the generated states from the explore operation may be in an arbitrary order and the merge requires sorted states to work effectively, the (large) buckets of states are sorted by the *sort* operation.

The last operation – *encode* – forms the final output stream in the network format and passes it to the output stack. It is a symmetric counterpart to the decode operation.

To achieve independence of the operations and thus low synchronization overhead, the states passed to the operations (i.e. intermediate results) are grouped into buckets. The operations on buckets can execute independently, synchronization occurs only when buckets that wait for processing are stored into or retreived from bucket buffers (gray boxes in Fig. 2).

Remaining to be described in Fig. 2 are the *input* and *output* stacks. The role of the input stack is to continuously receive the input stream and pass it to the *verification core* as required. We achieve this by keeping two separate buffers: the *network-side* buffer and the *input* buffer. When the size of a buffer reaches a predefined limit, a part of it is stored asynchronously on the external storage device (for the network side buffer) or retrieved from the external storage device (for the input side buffer). We choose the capacity of the buffers and the limits big enough to cover the access latency of the external storage device.

The *output stack* behaves in the same manner as the input stack. Its role is to fluently receive a newly created stream and pass it to the successor in the circle.

As far as scheduling is concerned, we have achieved the best results when using a strict priority policy. We take the operations in the order of their description in the text, test the availability of input data and execute the first operation whose input is available. If none of the operations can be executed, we temporarily block the worker until some of the buffers changes.

4 Experiments

This section presents the results we have obtained during the verification of several models using our prototype implementation.

We have chosen four models – Dining Philosophers, Producer-Consumer, an artificial parallel test and an industrial example from [14]. Dining Philosophers and Producer-Consumer are classical synchronization problems widely used for experiments. The artificial parallel test consists of pairs of handshaking components and is parameterized by the number of pairs. The state space of the test has an acyclic structure, high number of duplicit states and an upper bound on the degree of nodes. A diagram of the state space would look like a very complex diamond. The industrial example aggregates the tests from [14] to achieve a sufficiently large state space.

To get representative results, we have used the simplest configuration – a basic lexicography comparator and state space reductions. The experiments have been executed multiple times and the cases that exhibit significant variance of results are explicitly mentioned. The hardware characteristics of the nodes are in Tab. 1.

As the first experiment, we have measured the dependence of the verification time on the number of threads (Fig. 3, Tab. 2). We have used a single computer with system configuration 1 from Tab. 1. Increasing the number of threads speeds the verification up proportionally, as evidenced by the straight lines on the logarithmic-scale graph. From 1 to 8 threads, the speedup is close to optimal v/t, where v is the verification time and t is the number of threads. The break at 8 threads is caused by hyperthreading, which runs two threads concurrently on each processor core. Unfortunately, because parts of the core are shared, the performance increase is lower than when using a standalone processor core.

An experiment measuring the dependence of the verification time on the number of computational nodes is also very promising (Fig. 4, Tab. 3). For maximum clarity, we have used up to 8 uniformly configured computers with system configuration 2 from Tab. 1. The verification speed increases proportionally to the number of nodes, again evidenced by the straight lines on the logarithmic-scale graph. The average network bandwidth during the experiment was 344kB/s, which is significantly below the theoretical 100Mb/s throughput. Compared to the multiprocessing experiment, we can see a slowdown of about 5%. This overhead can be attributed to the network layer.

Figure 5 and Tab. 4 show the dependence of the verification time on the size of the input disk buffers. The experiment uses the Producer-Consumer test with 2 buffers, which generates a large state space that does not fit into memory and has to be stored on disk. To simulate potential user in-

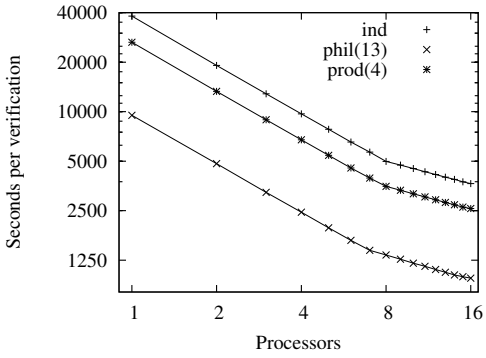


Figure 3. Scalability in the number of threads.

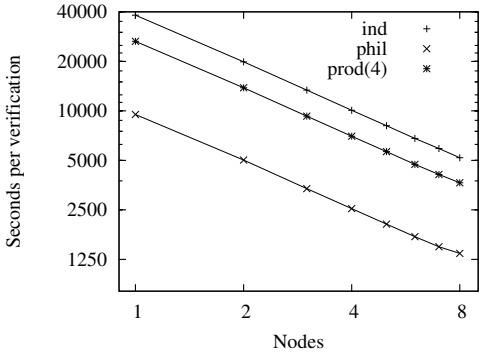


Figure 4. Scalability in the number of nodes.

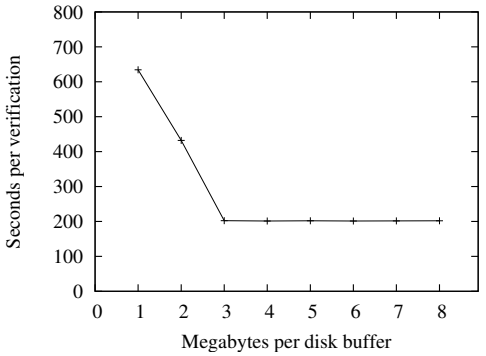


Figure 5. Dependece on the size of input disk buffers.

System configuration 1

Model	Dell PowerEdge 6850
Processor	4x Intel Xeon 7140M 3.40GHz 16kB L1, 2MB L2, 16MB L3, Dual-core, Hyperthreading
Chipset	Intel E8501, 800MHz FSB
Memory	32GB organized as 4x8x1GB, DDR2-400 PC2-3200 ECC
OS	RedHat Enterprise Linux v4 Advanced Server x86_64, Linux 2.6.9-42.0.3.ELlargesmp

System configuration 2

Processor	AMD Athlon 64 3000+ 1800MHz, 64kB L1, 512kB L2
Chipset	VIA KT800Pro
Memory	512MB organized as 2x256MB DDR 400MHz PC3200 CL2 (Dual Channel)
Network	Yukon Gigabit Ethernet card 100Mb
Hard disk	Seagate Barracuda ST3250823AS
OS	Debian GNU/Linux 3.1 Sarge, Linux vanilla 2.6.17.6

Network switch

Model	Cisco Catalyst WS-C2950G-48-EI/B0
IOS	12.1(22)EA9

Implementation

Language	C99
Compiler	gcc 3.3.5 (Debian 1:3.3.5-13)
Optimizations	`-O2`

Table 1. Reference systems

Test	**States**	**Verification time (mm:ss)/Number of threads**							
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
phil(13)	268 055 039	158:24	80:29	53:51	40:46	32:47	27:30	23:54	22:23
prod(4)	312 343 245	440:44	221:20	148:42	112:18	90:26	75:41	65:46	58:31
ind	423 514 367	635:45	318:19	214:05	161:35	130:10	108:52	94:36	83:04
		9	**10**	**11**	**12**	**13**	**14**	**15**	**16**
phil(13)	-	21:06	19:57	19:07	18:16	17:34	16:56	16:31	16:12
prod(4)	-	55:28	52:46	50:30	48:28	46:44	45:14	43:47	42:56
ind	-	78:50	75:07	71:52	69:04	66:38	64:34	62:23	60:54

Table 2. Scalability in the number of threads.

Test	**States**	**Verification time (mm:ss)/Number of nodes**							
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
phil(13)	268 055 039	158:27	83:42	56:01	42:24	34:05	28:37	24:51	22:38
prod(4)	312 343 245	440:58	230:11	154:40	116:49	94:04	78:44	68:24	60:52
ind	423 514 367	635:55	331:03	222:39	168:03	135:23	113:14	98:23	86:24

Table 3. Scalability in the number of nodes.

Test	**Verification time (mm:ss)/Buffer size in megabytes**							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
prod(2)	10:34	7:12	3:43	3:28	3:22	3:21	3:21	3:21

Table 4. Dependence on the size of input disk buffers.

terference, we have copied unrelated files on the disk during the experiment. Our results show that the minimum buffer capacity for effective use of the disk is 3MB. The results obtained with 1MB and 2MB buffers have significant variance and have been partially distorted by the kernel read-ahead feature.

In the last experiment, we have attempted to create as large a cluster as possible. We have pooled 47 desktop workstations within our department. The configurations vary from AMD Athlon 900MHz to AMD Athlon 64 3200+ at 2GHz. The cluster verifies the industrial example within 12 minutes, with the average speed of 597254 closed states per second. The example of 16 dining philosophers (21.10^9 states) has been enumerated within 11 hours.

5 Evaluation

The distinguishing feature of the proposed approach is sequential rather than random access to the states. The obtained results show that the method not only works but also scales well. The differences between the optimal (in the sense of ideal load distribution) and the measured values are relatively low. An important feature of the streaming method is its ability to adapt to non-uniform cluster with dynamic performance changes typical for user workstations. This is rarely achieved by hash-based approaches.

The difficult part of the traditional approach to distributed model checking is the choice of the state space partitioning function. Similar role (but not the same) within the streaming approach can be found in the choice of the state ordering function. With the traditional approach, poor partitioning function leads to poor distribution of workload among nodes. With the streaming approach, poor ordering function leads to higher distance in the stream of states that are neighbors in the state space and thus higher communication latency. Note, however, that unlike with the partitioning function, a poor choice of the ordering function does not lead to workload asymetry.

In general, a metric for the quality of the ordering function is the ratio of the distance in the stream to the distance in the state space, the smaller the better.

We have performed several comparison tests with the hash-based partition method on a symmetric cluster of uniform nodes. The hash-based implementation uses standard uniform hash function and message buffering to improve network utilization, while the streaming implementation uses basic lexicographic ordering. For medium-sized models, the hash method has generated lower network load, however, the performance of both methods has still been comparable. We have observed that in some cases, the generation of new states was up to 2 times faster in the streaming implementation than in the hash implementation, although both implementations shared the same code. This effect is due to memory caches working better when the state look-up tables are accessed more predictively, which is the case with the lexicographic state ordering. This speedup compensates the cost of merging new states into the sorted stream.

For dense models with many small cycles, surprisingly, the streaming implementation generates *lower* network load. Where the traditional approach has to send $O(m)$ messages where m is the number of transitions, the streaming approach saves the states in a buffer until the right place in the stream reaches the node, and thus eliminates many duplicates locally.

For large models and the traditional method, nodes start swapping when the size of the state tables exceeds the available memory. Consequently, the verification speed drops down to only a fraction of the original speed. In contrast, the streaming method uses disks to store the state space and the verification continues smoothly even when the stream does not fit into the available memory.

During the experiments with large models, the network bandwidth was not saturated. The stream flow is relatively slow and does not place high demands on the network infrastructure. Although the stream speed is determined by the fastest computer in the cluster, even very slow computers (below 10% of the fastest computer) did not represent a bottleneck. This is because stream forwarding has very low requirements compared to successor state generation, allowing the load balancing mechanism from Sect. 3.7 to compensate.

Even when the method produces larger network bandwidth, the streamed data transfers fit the contemporary hardware perfectly and are therefore processed more efficiently than random access to the states. Note that although the circulation of the closed states may appear inefficient at the first glance, it is typically not an issue. Since the exploration is close to BFS, the number of epochs is close to the depth of the traversed model, which tends to be small in real-life models [21].

The efficiency of the approach also depends on the state space structure. For example, a state space that consists of a single path will be not traversed effectively. Fortunately, similar situations are very rare as state spaces tend to branch a lot and describe many concurrent executions. When necessary, the model can also be adjusted before the verification in order to reduce the presence of such pathological constructions.

6 Related work

Distributed verification is heavily studied and several significant improvements to existing model checkers have been created so far. The approximation of an optimal partitioning function was addressed in several ways. Among

the first proposed methods was the use of a hash function in Stern and Dill's [25], where the distributed version of the Murφ verifier was described. The table of states is partitioned across the computational nodes, which allows addressing more states than on a single node and carry the verification out in parallel. While this approach divides the states evenly, it does not reduce the number of cross-border transitions. The method has been improved by exploiting common constructs in the model (for example Lerda and Sisto's [17] by relying on one state component only). Another attempt was based on interpolating partial state space obtained by pre-exploration of the state space [20]. Additionally, to mitigate the problems associated with imperfect partitioning, model verifiers were proposed to periodically check the current load and redistribute the load by adjusting borders on the fly [7].

While some approaches target safety properties, there are also approaches for verifying liveness properties expressed mainly in linear temporal logic (LTL). Since LTL model checking relies on finding accepting cycles in the state space by the nested DFS, and DFS is inherently sequential [24], distributed LTL model checking is complicated by keeping additional information and enforcing synchronization. Barnat, Brim, and Stribrna proposed a distributed variant [5] of the SPIN model checker [12]. The method is based on updating dependency structure of all accepting states and boundary-crossing transitions. Nested DFS parts are then processed sequentially for all the accepting states according to the order specified by the dependency structure. Jones and Sorber in [15] proposes a new parallel algorithm for LTL model checking using coordinated depth-bounded random walks.

State space splitting for distributed model checking has been used also for the computation tree logic (CTL) in a work of Brim, Yorav, and Zidkova [8]. Each computational node verifies an incomplete part of the state space, modelled as a Kripke structure. Nodes exchange assumptions about formula values via border states.

Stern and Dill have also proposed an external model checking in Murφ [26]. An improved version was presented by Bao and Jones [4]. They propose a variant of parallel partitioned hash table algorithms and use a chained hash table. There are several approaches to external BFS [10, 19]. In [16], Korf and Schultze use a perfect hash function for the delayed cycle detection on an external storage drive, but the distribution was not addressed.

An improvement to delayed cycle detection has been proposed in [27] by Zhou and Hansen. Unfortunately, the method aims at graphs with a regular structure which is not typical in model verification. Jabbar and Edelkamp in [13] extend the External A* [9] algorithm for parallel model checking. The approach is based on observation that the internal work on each state bucket can be parallelized on more processors. The method is distributed and parallel, unfortunately presented for 3 processes only.

7 Conclusion

We have introduced a novel method of distributed model verification of safety properties for large state spaces, proposed as an alternative to traditional partitioning-based approaches. The method is targetted at common network infrastructures of non-dedicated workstations.

Preliminary results show that the method scales very well with the number of processors and computational nodes involved. When the state space exceeds available memory, it can be temporarily stored on external storage devices with very low overhead. Additionally, the streaming method fits very non-uniform clusters and allows to balance the load dynamically during the verification. The ordering function between states can be constructed in order to address state locality, although the method does not rely heavily on this.

Acknowledgments.

The authors are grateful to anonymous reviewers for their valuable comments. This work was partially supported by the Grant Agency of the Czech Republic project 201/06/0770.

References

[1] J. Adamek. Addressing unbounded parallelism in verification of software components. In *Seventh ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2006)*, Las Vegas, Nevada, USA, Jun 2006. IEEE Computer Society.

[2] J. Adamek and F. Plasil. Erroneous architecture is a relative concept. In *Software Engineering and Applications (SEA) conference*, pages 715–720, Cambridge, MA, USA, Nov 2004. ACTA Press.

[3] A. Aggarwal, R. J. Anderson, and M.-Y. Kao. Parallel depth-first search in general directed graphs. In *STOC '89: Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 297–308, New York, NY, USA, 1989. ACM Press.

[4] T. Bao and M. Jones. Time-efficient model checking with magnetic disk. In N. Halbwachs and L. D. Zuck, editors, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 526–540. Springer, Apr 2005.

[5] J. Barnat, L. Brim, and J. Stribrna. Distributed LTL model-checking in SPIN. In *8th international SPIN workshop on Model checking of software*, pages 200–216, New York, NY, USA, 2001. Springer-Verlag.

[6] G. Behrmann, K. G. Larsen, and R. Pelanek. To store or not to store. In *Proceedings of Computer Aided Verification*,

volume 2725 of *Lecture Notes in Computer Science*, pages 433–445. Springer-Verlag, 2003.

[7] V. Braberman, A. Olivero, and F. Schapachnik. On-the-fly workload prediction and redistribution in the distributed timed model checker Zeus. *PDMC: 3rd International Workshop on Parallel and Distributed Methods in Verification*, 128(3):3–18, May 2005.

[8] L. Brim, K. Yorav, and J. Zidkova. Assumption-based distribution of CTL model checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(1):61–73, 2005.

[9] S. Edelkamp, S. Jabbar, and S. Schrodl. External a*. In S. Biundo, T. W. Fruhwirth, and G. Palm, editors, *KI 2004: Advances in Artificial Intelligence, 27th Annual German Conference on AI*, volume 3238 of *Lecture Notes in Computer Science*, pages 226–240. Springer, 2004.

[10] H. Garavel, R. Mateescu, and I. Smarandache. Parallel state space construction for model-checking. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 217–234, New York, NY, USA, 2001. Springer-Verlag.

[11] V. Holub. On verification of generalized interaction models of software components. In *European Conference on Object-Oriented Programming: Doctoral Symposium*, Jul 2006.

[12] G. J. Holzmann. The model checker SPIN. *Transactions on Software Engineering (TSE)*, 23(5):279–295, Jan 1997.

[13] S. Jabbar and S. Edelkamp. Parallel external directed model checking with linear I/O. In E. A. Emerson and K. S. Namjoshi, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 237–251. Springer, Jan 2006.

[14] P. Jezek, J. Kofron, and F. Plasil. Model checking of component behavior specification: A real life experience. In *International Workshop on Formal Aspects of Component Software (FACS'05)*, volume 160, pages 197–210, Aug 2006.

[15] M. D. Jones and J. Sorber. Parallel search for LTL violations. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(1):31–42, 2005.

[16] R. E. Korf and P. Schultze. Large-scale parallel breadth-first search. In M. M. Veloso and S. Kambhampati, editors, *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference*, pages 1380–1385. AAAI Press AAAI Press, The MIT Press, 2005.

[17] F. Lerda and R. Sisto. Distributed-memory model checking with spin. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 22–39, London, UK, 1999. Springer-Verlag.

[18] M. Mach, F. Plasil, and J. Kofron. Behavior protocol verification: Fighting state explosion. In *International Journal of Computer and Information Science*, volume 6, pages 22–30. ACIS, Mar 2005.

[19] K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *ESA '02: Proceedings of the 10th Annual European Symposium on Algorithms*, pages 723–735, London, UK, 2002. Springer-Verlag.

[20] D. M. Nicol and G. Ciardo. Automated parallelization of discrete state-space generation. *Journal of Parallel and Distributed Computing*, 47(2):153–167, Dec 1997.

[21] R. Pelanek. Typical structural properties of state spaces. In *Proceedings of SPIN Workshop*, volume 2989 of *Lecture Notes in Computer Science*, pages 5–22. Springer-Verlag, 2004.

[22] D. Peled. Combining partial order reductions with on-the-fly model-checking. In D. L. Dill, editor, *Proceedings of the 6th International Conference on Computer Aided Verification*, volume 818 of *Lecture Notes In Computer Science*, pages 377–390. Springer-Verlag, Jun 1994.

[23] F. Plasil, S. Visnovsky, and M. Besta. Bounding component behavior via protocols. In *TOOLS*, volume 30, pages 387–398, USA, Aug 1999. IEEE Computer Society.

[24] J. H. Reif. Depth-first search is inherently sequential. In *Information Processing Letters*, volume 20, pages 229–234, 1985.

[25] U. Stern and D. L. Dill. Parallelizing the murphi verifier. In *9th International Conference on Computer-Aided Verification (CAV)*, pages 256–278. Springer-Verlag, 1997.

[26] U. Stern and D. L. Dill. Using magnatic disk instead of main memory in the murphi verifier. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 172–183, London, UK, 1998. Springer-Verlag.

[27] R. Zhou and E. A. Hansen. A breadth-first approach to memory-efficient graph search. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference*. AAAI Press, Jul 2006.

Chapter 9

Identifying Representatives for Interfering Automata

Viliam Holub

Contributed paper at **1st International Conference on Digital Communications and Computer Applications** [1].

In conference proceedings,
published by the Jordan University of Science and Technology,
pages 1178–1185,
ISBN 625-3-2007,
2007.

The original version is also available electronically from the conference's site at http://www.cis.just.edu.jo/dcca2007/.

Identifying Representatives for Interfering Automata

Viliam Holub
Charles University, Czech Republic
holub@dsrg.mff.cuni.cz

ABSTRACT

In model checking, the number of states of a model tends to grow exponentially with the size of the model's description, leading to unacceptable space and time requirements. Focusing on generalized interaction protocols (Interfering Automata), we present a method of substitution of groups of states by a single state (representative) during the verification. As the number of representatives is significantly smaller than the size of the whole state space, our method pushes the limits of practical verification. Our approach is focused on parallel and distributed model checking.

Key Words: formal verification, model checking, partial order reduction, state explosion

1. Introduction

As the complexity of software systems and the cost of the failure grow, the industry starts using formal verification to discover errors. A software system is expressed as a formal model and tested for specified properties using a model checker. A naive implementation of a model checker generates and tests all the states of the model. Unfortunately, the number of states tends to grow exponentially with the size of the model and the verification process quickly reaches practical limits of memory and time consumption (the state space explosion problem).

The state space size is determined basically by the degree of parallelism obtained in the model. This is illustrated in Fig.1 which shows two parallel processes passing through a sequence of states. Searching for deadlocks in a system of two independent (with respect to communication) processes leads to visiting *nxm* states, while only *n+m* states have to be visited - a significant difference. The key idea is to find those states that are crucial for obtaining a correct result and omit the superfluous ones resulting in a state space reduction - we say that those key states are representative states (or *representatives* for short).

Our goal is to find representatives during the verification of a model expressed in the formalism of Interfering Automata in a way suitable for distributed verification. Interfering Automata (first mentioned in [1]) is a formal language which allows simulating several Label Transition System (LTS)-based formalisms. The design of Interfering Automata is focused on a fast automatic exploration and its typical use is as an intermediate language during the verification process.

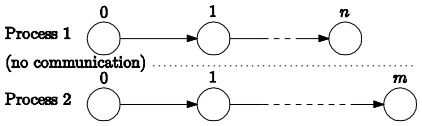


Fig. 1. An example of inefficiency: a simple checker searching for all deadlocks would generate all the possible interleaving of two independent processes.

We first summarize the necessary background in Sect. 2. Along with a precise definition of Interfering Automata, we briefly explain a practical modeling language called Behavior Protocols[2] which we use for examples, and explain the process of state space exploration. The method is explained in Sect. 3 - we identify the notion of a reduction, analyze an approach of constructing representatives, attempt to further improve the method by heuristics, and proof the correctness of the method. How the reduction method works in prac-

tice is discussed in Sect. 4. The idea of state space reduction has been studied by many researches. In Sect. 5, we present alternative and related approaches. Finally, our work is concluded in Sect. 6.

2. Background.

2.1 Behavior Protocols

Behavior Protocols[2] is a textual description of communication among software components, describing all allowed scenarios of message passing on component boundaries. We use Behavior Protocols for examples, because, in comparison with temporal logics, Behavior Protocols are easier to comprehend on the first look.

We can express two types of actions - a non-blocking *emit a message* (denoted by !) and a blocking *accept a message* (denoted by ?). A message is identified by an interface name and a method name, and an action type (↑ for request and ↓ for response). For example, !i.a↑ is emitting a request a on an interface i, and ?r.b↓ accepting a response b on an interface r. A message flow is expressed by operators - sequencing ;, alternative +, repetitive *, and an arbitrary interleaving |. For example, a call of a function m1 or m2 on an interface I would take the form (!i.m1↑; ?i.m1↓)+(!i.m2↑; ?i.m2↓).

Component protocols are combined with a *composition*[2] operator. The semantics of a composition is to decide whether components cooperate without problems, i.e. searches for emitting a message without an accepting counterpart (a *bad activity* error) and deadlocks (a *no activity* error).

2.2 Interfering Automata

Many formal languages based on LTS already exist - theoretical Interface Automata[3], Team Automata[4], and Component-Interaction Automata[5] and those based on Architecture Description Language (ADL) like Behavior Protocols[2], and Darwin[6]. These languages allow specifying the behavior of a single component and define relations among them to check the properties like equivalence[7], compatibility[3], and compliance[8]. In contrast, an automaton in Interfering Automata is just a set of states where exactly one is active, and the behavior is defined on a group of automata only. To emphasize this, we refer to the automaton as a *participant* and to a set of interacting participants as a *system*.

In contrast to traditional approaches, interfering automata are not intended to be a language for direct modeling by humans. Its objective is to be easily explorable by tools and allows simulating other LTS-based formalisms. There is no limitation to the type of neither synchronization (synchronous, asynchronous or blocking, non-blocking) nor the number of action participants (one-to-one, one-to-many, and many-to-many).

In the following text, we will keep a notation of the i-th element of a vector as [i], and P(a) represents the set of all subsets of a.

Interfering automata are formally defined as a triple (n, N, Δ), where n is total number of participants, a vector N specifies the number of states of all participants, and Δ is the set of transition templates.

A *system state* S is a vector of active states of all the participants, and thus for each $i = 1..n : 0 \le S[i] < N[i]$. A *starting state* is a special instance of a system state where all the active participant states are equal to 0. Thus the starting state takes the form of the zero vector $(0,...,0)$.

A *transition template* is a pair $a = (C, R)$ where:

- *condition* C is a vector which specifies the system states the transition template may be applied to, and $\forall i = 1..n : C[i] \in P(\{0..N[i]-1\}) \setminus \varnothing$, is a set of allowed participant states. For example, a condition $(\{1,3,8\},\{5\})$ specifies that the first participant must be in states 1, 3 or 8 and the second participant must be in the state 5.
- *result* R is a vector, where $\forall i = 1..n : R[i] \in P(\{0,..,N[i]-1\})$. Each element of the result specifies a new value of participant states. Multiple states ($|R[i]| \ge 1$) are interpreted as a non-deterministic choice,

while empty states ($R[i]=\emptyset$) are interpreted as a ``keep untouched" rule and the state of the participant i is not to be changed.

A system state s *meets* the condition of a transition template $a=(C,R)$ if and only if $\forall i=1..n:\ s[i]\in C[i]$.

We say that a transition template a is *enabled* in a system state s, if s meets the condition of a. Otherwise, we say that a is *disabled* in s.

Applying a transition template $\alpha=(C_\alpha,R_\alpha)$ to a system state s results in a system state z such that for $\forall$i=1..n: $z[i]=s[i]$ if $r[i]=\emptyset$ and $z[i]=x,\ x\in r[i]$ otherwise. The application of a transition is denoted as $z=a(s)$ and $s\xrightarrow{a}z$.

To improve readability, we will follow these convention rules of how to write transition templates:

- the condition and result parts of a transition are separated by an arrow $\rightarrow$ and both embedded within angle brackets $<...>$,
- empty state sets are omitted and sets with only one element are written without curly brackets {...},
- in the condition part, sets that contain all the participant states are omitted as well (note that empty sets are forbidden in conditions),
- prefixing states with the exclamation mark ! means ``all but".

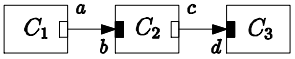
For illustration, $<1,\{2,3\},!4>\rightarrow<2,\{6,7\},,>$ is interpreted as a transition which could be applied to all the system states which have the first participant in a state number 1, the second participant must be in states number 2 or 3, and the third participant must not be in a state 4. The two system states resulted from applying the transition will have the first participant in a state 2 and the second one in a state 6 or 7 respectively. The transition does not change the state of the third and fourth participant.

A *path* p from a state s to the state z is a sequence of system states s_j and transitions e_j such that $p=s\xrightarrow{e_1}s_1\xrightarrow{e_2}\cdots\xrightarrow{e_n}z$ and for $\forall i=1..n:\ s_i=e_i(s_{i-1})$.

A state z is *reachable from* s if there exists a path from s to z. If a state s is reachable from the starting state, we say that s is *reachable*.

A system state s is *deadlocked* (s is a *deadlock*) if all transition templates are disabled in s.

System layout



Behavior specification

C_1 $!a.m1\uparrow;?a.m1\downarrow$

C_2 $?b.m1\uparrow;!c.m2\uparrow;?c.m2\downarrow;!b.m1\downarrow$

C_3 $(?d.m2\uparrow;!d.m2\downarrow)*$

LTS

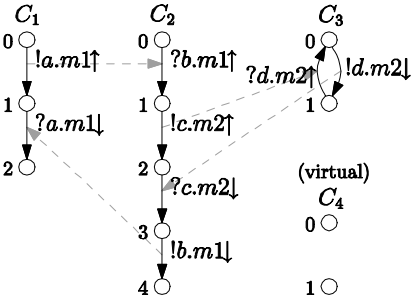


Fig. 2. Example of a system layout, behavior description, and LTS transcription.

A reachable deadlock is crucial - all important checked conditions must be presented as a deadlock in the model. This is not limiting us because of the power of transition specification. For example, Fig. 2 shows a system of three components and their specification in Behavior Protocols. The composition operator of these three components in Interfering Automata would take the form:

$$\begin{array}{cc} & A \\ <0,0,\_,0> & \rightarrow <1,1,\_,\_> \\ <\_,1,0,0> & \rightarrow <\_,2,1,\_> \\ <\_,2,1,0> & \rightarrow <\_,3,1,\_> \\ <1,3,\_,0> & \rightarrow <2,3,\_,\_> \end{array}$$

B

$< 0, !0, \_, 0 > \rightarrow < \_, \_, \_, 1 >$

$< \_, !1, 0, 0 > \rightarrow < \_, \_, \_, 1 >$

$< \_, !2, 1, 0 > \rightarrow < \_, \_, \_, 1 >$

$< !1, 3, \_, 0 > \rightarrow < \_, \_, \_, 1 >$

The match of emit-accept actions are expressed by transitions *A*. Transitions *B* express a bad activity error by enforcing the model to reach deadlock (*failure transitions*). This is achieved by traversing to the state 1 of the fourth (additional) participant and requesting the state to be 0 for all the transitions.

2.3 State Space Exploration

State space exploration (or simply *exploration*) is a process of systematic traversal of system states. The goal is to decide which specified states are reachable in the model and report them to the user. The notion of a state we are looking for varies among different formal models. It includes such attributes as a fail of a condition, deadlock, and reachability from the starting state.

We will describe a simple general exploration algorithm which is satisfactory for our needs. Three sets of states are used - a set of visited but not explored states *V* (we have not generate their successors yet), a set of explored states *E* (successors have already been generated) and a temporary set of states *S*. The exploration process begins in the starting state (see the algorithm below) and proceeds by repetitive application of an *exploration step* (operations within the repeat loop).

- Set initial values E=0, V={starting state}, S=0

repeat

- Take one state s from V
- Generate all successors of s to S
- Report s to the user if interesting (s is deadlocked if and only if S=0)
- Remove already explored states (in E) and the state s from S
- Add s to explored states E
- Add S to visited states V

until V=0

This algorithm visits all the reachable states, which are eventually included in *E*. A set of all reachable states is referred to as a *full state space*. If some of the states have been omitted due to optimizations, we say that the set is a *reduced state space*. Exploring algorithms differ mostly in data structures used and in the scheme of how the states ``to explore" are chosen. Most often used instances of this algorithm are referred to as depth first search (DFS) and breadth first search (BFS).

3. Identifying representatives

3.1 Representatives

Our goal is to reduce the state space that has to be explored, without loss of the ability to report important reachable states. In other words, by applying reducing optimizations we must not lose any deadlocked state. Because the size of the reduced state space is smaller than the full state space, states contained in the reduced state space may *represent* (be *representative* of) more than one state in the full state space.

In a single exploration step, the model checker has to decide either whether some of the newly explored states are represented by some of the already explored states or whether there is a relation of representativity among newly explored states themselves. In this particular moment, the problem can be converted from the relation between states to the relation between two enabled transitions - derivately by considering the resulting states: a transition a is a representative for a transition b in a state s if and only if $a(a)$ is a representative for $b(s)$.

Let's analyze the exploration step in a general system state s. Our task is to find a *representative* subset d from the set of all enabled transitions in s. Thus, every enabled transition in s is in d, or is represented by some transition from d. Furthermore, only this subset of transitions will be used for the exploration step.

We can iteratively test all the pairs of enabled transitions and remove those which

already have a representative. This simplifies the problem to finding a relation between two transitions a and b, which says whether a is a representative of b.

Given that all the properties that we want to check are expressed by (the) reachability of a deadlock state, we can say that a transition a is a representative for a transition b if there is no deadlock reachable through b that would not be reachable through a.

3.2 Sensitive transitions

Suppose that the deadlocked state d is reachable from s by a path: $s = s_0 \xrightarrow{e_1=b} s_1 \xrightarrow{e_2} \cdots \xrightarrow{e_n} s_n = d$ and not reachable from $a(s)$ (Fig. 3). We know that a is enabled in s, but is not enabled in d. This means that there exists such e_i: $s_{i-1} \xrightarrow{e_i} s_i$ that a is enabled in s_{i-1}, but is not enabled in s_i. The transition e_i incorporates changes that violate a conditions. Because it is not easy to decide from s which transitions will break the conditions of a, we will introduce a slightly modified relation a is *sensitive* to e_i and further assume that the relation holds in e_1.

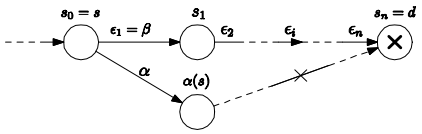


Fig. 3. Modeling situation: a path from a state s to a deadlocked state d. State d is reachable from s by applying b first, but is not reachable by applying a first.

Formally, a transition template $a = (C_a, R_a)$ is *sensitive* to transition template $e = (C_e, R_e)$ in a state s, if a is enabled in s and for at least one participant i the following holds:

- the participant state $s[i]$ meets the condition of e, i.e. $s[i] \in C_{e[i]}$,
- a does not allow all the states of participant i, i.e. $C_a \neq \{0..N[i-1]\}$,
- the result part $R_e[i] \neq \emptyset$ and $R_e[i] \neq s[i]$.

We can simply avoid marking a as a representative if it has a sensitive transition. Although this is a sufficient restriction which guarantee the correctness of the exploration, we are loosing some representatives due to the fact that not all transitions which a is sensitive to are reachable from s.

3.3 Reachability of sensitive transitions

Given a situation from Sect. 3.2, our aim is to decide whether a state s_{i-1} (and, derivately, a transition e_i) is reachable from the state s ($b(s)$). Unfortunately, we are solving the problem of reachability itself, thus the precise calculation would not help us. This is a common problem of reductions based on partial order reduction (POR). As already shown by many researchers ([9] for example), the problem is at least as hard as the model checking itself.

Rather than finding the best possible solution, we have to consider the representative relation as heuristics - to find an approximation under acceptable cost during the verification. While the false-negative error is unacceptable, the false-positive error does not break the correctness, but declines the efficiency of the reduction. The power of heuristics is crucial and varies among different approaches. For our purposes, we will heuristically decide whether the selected sensitive transition is also reachable.

Although originally introduced for state compression, we will use a relation of *dependance* described in [10]. The idea behind state dependance is an observation that a participant active state strongly depends on active states of other participants. Formally, the relation of dependance takes the form $((p_1, r_1), (p_2, r_2))$ where p_1 and p_2 are participants and r_1 and r_2 are respective participants' states. The relation $((p_1, r_1), (p_2, r_2))$ does not hold if a participant p_1 is active in a state r_1 and partici-

pant p_2 cannot be active in state r_2. Otherwise, the relation holds.

We can test the reachability of e_i from s by trying to create an unknown state s_{i-1}. Because a and e_i are both enabled in s_{i-1} and a is sensitive to e_i in s, we have enough information to construct candidates for s_{i-1} by restricting possible state values. Let's initialize s_x such that for $\forall j = 0..n$: $s_x[j] = s[j]$ if $C_a[j] \neq \{0..N[j]-1\}$, and $s_x[j] = C_{e_i}[j]$ otherwise. Then for all participants j, all participants $p \neq j$ and all the states k in $s_x[p]$ test the dependance $((j, s[j]),(p,k))$. If it does not hold, remove k from $s_x[p]$.

After all the iterations, s_x holds all the candidates for s_{i-1}. If there exists a participant p such that $s_x[p] = \varnothing$, there is no suitable candidate for s_{i-1} and e_i is unreachable from s.

3.4 Proof of correctness

Let's enrich the algorithm from Sect. 2.3 with constructing representatives as described in Sect. 3.2 and 3.3. Suppose that the algorithm does not report a deadlock state d reachable by a path: $(0,\ldots,0) = s_0 \xrightarrow{e_1} s_1 \xrightarrow{e2} \cdots \xrightarrow{e_n} s_n = d$. There must exists a state s_i such that the states $s_0,\ldots,s_{i+1}$ have been visited except s_{i+1}. This means that in a state s_i the algorithm has chosen some a as a representative for e_{i+1}. Thus the algorithm

- does not find a sensitive transition in s_i. It follows from the definition of the sensitive relation that a was enabled after all successive $e_{i+1},\ldots,e_n$ and thus in a state d. But this is not possible, because d is deadlocked.
- does find a sensitive transition $e_j : j \geq i$, but marks it as unreachable. A transition e_j was enabled in all the initial candidate states s_{j-1}, but not after iteratively removing participant states based on state dependance. Thus, a reachable state has been removed. Assuming that the relation of dependance is correct, this is not possible.

4. Experiments

A prototype has been implemented to demonstrate the techniques described in this paper. To evaluate our approach, we present the results of two examples (Tab. 1). The first one analyzed is synthetic (theoretical). It consists of a composition of component pairs, each pair invokes a method call[11]. This involves a high parallelism of independent component pairs. As shown in the table, the parallelism has been completely removed by the relation of sensitivity.

Test name	FSS size	RSS sensitivity	RSS sensitivity and reachability
Parallel test[11]	1048576	40	40
CS1[12]	10458	10458	56
CS2[12]	635780	126762	3123
CS2[12] SPIN	17128	-	17128

Tab. 1. Results of constructing representatives. FSS stands for a full state space, RSS for a reduced state space, and CS for a case study.

Based on a real case-study[12], the second example CS1 is a composition of selected components "arbitrator". The sensitive relation does not reduce the state space at all. This is caused by a high communication among components and a lot of failure transitions. However, when the failure transitions have been effectively removed by the relation of reachability, the state space is reduced by the factor of 180.

The third example CS2 contains the whole case-study. The sensitive relation reduces the state space significantly, supposedly due to the optimized asynchronous timer that involves a high degree of parallelism. The reachability relation deflates the state space even more.

Generally, the relation of sensitivity fails in situations where there are more than one method implementations in the protocol,

where there is an alternative (+) of a control flow, and in situations of a lot of unoptimized unreachable transitions in the model.

The fourth example compares our method with the state-of-the-art model checker SPIN. The specification of CS1 has been translated to the Promela language and verified with enabled and disabled partial order reduction. To our surprise, SPIN has serious problem to optimize the verification in this type of the model. Although the partial order reduction does change the behavior of the checker - reflected by the printed statistics - the total number of stored states does not change. We believe this is due to the "powerful" semantic of transitions that does not fit to the SPIN partial order reduction implementation and thus SPIN cannot be used effectively for the verification of such models.

5. Related work

A reduction of a state space has been studied by many researchers in many different scopes. The partial-order reduction technique[15][13][14] (along with BDD) seems to be the most efficient. In contrast with [15], our relation of representative is not equivalence. While being close to *ample sets*[9] in the exploration step, our approach better fits to the environment of general state exploration[16]. Also, our method better situates to distributed computing - to make the decision of sensitivity and reachability, we do not need to check the status of the other (possibly not locally stored) states and, furthermore, there is no need for reopening of already explored states which would lead to the problem of race-conditions.

A model checker for Behavior Protocols[11] generates the automata on-the-fly, while for Interfering Automata, participants have to be extracted before the state space exploration itself. This allows analyzing all the transitions in the exploration step and does not suffer any limitation as the sizes of automata (participants) are small.

6. Conclusion

We have presented a state space reduction by identifying representatives on-the-fly during the exploration. The key benefits of the method are:

- fits perfectly for distributed verification, because statuses of "near" states are not required for the reduction,
- is able to reduce the state space even in models where other approaches fails (Sect. 4),
- evaluated results shows that the method reduces the size of the state space significantly, and
- brings low overhead to the exploration process.

Acknowledgments. The author is grateful to Jiri Adamek and Petr Tuma for their valuable comments. This work was supported by the Czech Science Foundation (GACR) grant no. 201/05/H014 and 201/06/0770.

References:

[1] V. Holub, "On verification of generalized interaction models of software components", ECOOP: Doctoral Symposium, 2006

[2] F. Plasil, S. Visnovsky, and M. Besta, "Bounding component behavior via protocols", TOOLS, Vol. 30, USA IEEE, 1999, pp. 387 - 398

[3] L. de Alfaro and T.A. Henzinger, "Interface automata", FSE, New York, USA, ACM Press 2001, pp. 109 - 120

[4] C. Ellis, "Team automata for groupware systems", SIGGROUP, ACM Press, 1997, pp. 415 - 424

[5] B. Zimmerova, L. Brim, I. Cerna, and P. Varekova, "Component-interaction automata as a verification-oriented component-based system specification", SIGSOFT, Notes 31, 2006

[6] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, "Specifying distributed software architectures", ESEC,

London UK, Springer-Verlag, 1995, pp. 137 - 153
[7] J.R. Allen, “A formal approach to software architecture”, PhD thesis, CMU, 1997
[8] J. Adamek and F. Plasil, “Component composition errors and update atomicity: Static analysis”, SME Research and Practice, Vol. 17, No. 5, 2005, pp. 363 - 377
[9] D. Peled, “Combining partial order reductions with on-the-fly model-checking”, ICCAV, Vol. 818 LNCS, Springer-Verlag, 1994, pp. 377 - 390
[10] V. Holub, “State Dependances in Interfering Automata”, WDS, Matfyzpress, 2006
[11] M. Mach, F. Plasil, and J. Kofron, “Behavior protocol verification: Fighting state explosion”, Journal of CIS, Vol. 6, ACIS, 2005, pp. 22 – 30
[12] P. Jezek, J. Kofron, and F. Plasil, “Model checking of component behavior specification: A real life experience”, FACS, 1995, Vol. 160, pp. 197 – 210
[13] A. Valmari, “A stubborn attack on state explosion”, CAV, Vol. 531, LNCS, Springer-Verlag, 1991, pp. 156 – 165
[14] L. Brim, I. Cerna, P. Moravec, and P. Simsa, “Distributed partial order reduction of state spaces”, STC, Vol. 128, No. 3, 2005, pp. 63 – 74
[15] D. Peled, “Partial order reduction: model-checking using representatives”, MFCS, LNCS, Springer, 1996, pp. 93 – 112
[16] D. Bosnacki, S. Leue, and A.L. Lafuente, “Partial-order reduction for general state exploring algorithms”, MCS SPIN, Vol. 3925 LNCS, Springer, 2006, pp. 271 – 287

Chapter 10

Implementation of a Linux Log-Structured File System with a Garbage Collector

Martin Jambor

Contribution paper in **ACM SIGOPS Operating Systems Review** [8].

Published by ACM Press, New York, NY, USA,
pages 24–32,
ISBN 0163-5980,
2007.

The original version is also available electronically from the publisher's site at http://doi.acm.org/10.1145/1228291.1228299.

Implementation of a Linux Log-Structured File System with a Garbage Collector

Martin Jambor
jamborm@matfyz.cz

Jan Tauš
pan_tau@matfyz.cz

Tomáš Hrubý
byjac@matfyz.cz

Kuba Krchák
gkg@matfyz.cz

Viliam Holub
holub@nenya.ms.mff.cuni.cz

Department of Software Engineering
Charles University

ABSTRACT

In many workloads, most write operations performed on a file system modify only a small number of blocks. The log-structured file system was designed for such a workload, additionally with the aim of fast crash recovery and system snapshots. Surprisingly, although implemented for Sun Sprite and BSD systems, there was no complete implementation for the current Linux kernel. In this paper, we present a complete implementation of the log-structured file system for the Linux kernel, which includes a user-space garbage collector and additional tools. We evaluate the measurements obtained in several test cases and compare the results with widely-used ext3.

Categories and Subject Descriptors

D.4.3 [**Operating Systems**]: File Systems Management—*file organization*; D.4.2 [**Operating Systems**]: Storage Management—*garbage collection*; E.5 [**Data**]: Files—*organization/structure*

Keywords

Log-structured file systems, Linux file systems, garbage collection

1. INTRODUCTION

As random access memory is getting cheaper and more abundant in both personal computers and servers, many more workloads fit entirely into the disk cache. With most of the read requests satisfied by the cache, it is reasonable to optimize file systems primarily for writing. Moreover, quick crash recovery is a common requirement for a production file system and the most common way of achieving it is *journaling*. On the other hand, this technique incurs a write performance penalty[6] because it needs to write a portion of data twice to achieve a consistent metadata state at any given moment[13].

Log-structured file systems (LFS) have been proposed in [7] and first implemented for Sun Sprite system[9] in order to address these two issues. A log-structured file system writes all new information to a sequential structure referred to as the log, thus minimalizing the number of seeks and allowing fast crash recovery. They can also provide additional functionality not easily implemented by traditional file systems, such as snapshots.

Following the Sprite-LFS mentioned above, Seltzer et al.[10] implemented a LFS system for contemporary BSD systems in 1993. Unfortunately, over the course of time it has been removed from FreeBSD and OpenBSD. It is still present in NetBSD but it appears to be no longer completely functional as of NetBSD 2.0.2[14]. There have been several attempts to write LFS for Linux, but all except one have been abandoned without achieving their goals. The only exception is the currently developed NILFS project[5], but it still lacks working garbage collector which is a vital part of any LFS and also the part that poses most implementation issues. In the end of the day, there has not been an implementation of a traditional[1] LFS for an open-source operating system until now.

In this paper, we present a design and an implementation of LFS for Linux 2.6 kernel which takes full advantage of the page cache, has a working garbage collector, uses sophisticated data structures for large directories that considerably speed up directory operations, implements snapshots and is capable of fast recovery from a system failure. We concentrate on those parts that differ from the BSD implementation[10], how the file system is integrated to the current Linux environment and our solutions to the problems encountered during the implementation of the garbage collector and the segment management in general. We have also done a series of measurements to compare our file system with *ext3*.

We start with an overview of our implementation (Sect. 2), outlining the basic structures and describing the writing process and recovery mechanisms. The most significant issues that have arisen during the implementation are discussed in Sect. 3. This includes the free space and segment man-

[1]There are LFS for flash-based devices but they pursue different goals and are not considered by this paper.

agement and the garbage collector in particular. We evaluate the file system performance using several benchmarks in Sect. 4. Finally, we conclude the paper in Sect. 5.

2. IMPLEMENTATION OVERVIEW

Although the fundamental principle of LFS may seem simple, it presents us with two important issues. The first is a need for an indexing structure so that read requests can be performed without sequentially scanning the disk. We have adopted the traditional approach of representing files and directories with inodes and use the well known mechanism of direct and indirect blocks to quickly locate requested data[12]. We write both structures to log whenever changed.

The trickier problem is how to manage free space so that writes are coalesced into large contiguous bursts. As all other LFS, we have solved this issue by dividing the disk into fixed size segments[9, 10], one megabyte each. When processing a write request, LFS must find an empty segment first. Afterwards, it accepts write requests from the memory management, VFS layer or the garbage collector and keeps writing the given data to the allocated segment as long as the currently written entity fits in. When it does not, we finish the full segment by appending metadata (described below) and allocating and writing a new segment from that point on. Thus, we always write the current segment sequentially from the beginning to the end. The drawback of this approach is that we must copy all live data out of the segment before rewriting it.

LFS systems need to finish the current segment also after they have flushed dirty data during `sync` or `fsync` system calls. Moving on to a new segment would potentially waste a lot of disk space. Therefore, these file systems introduce the so called *partial segments*. Each physical segment consists of one or more partial segments (Fig. 1). Multiple partial segments usually result from the system calls mentioned above. Detailed description can be found in the official documentation of the project[2]. Each partial segment contains the following elements:

1. Data blocks and indirect blocks of files and directories.
2. Inodes
3. Journal enabling the roll forward utility to deal with directory operations (Sect. 2.3).
4. File info structures required to identify all data and indirect blocks so that the garbage collector can recognize live data (Sect. 3) and the roll-forward utility understands which data have been changed since the last checkpoint (Sect. 2.3).
5. Segment summary which contains information global to the partial segment, including but not limited to checksums, addresses and sizes of the individual entity blocks described above, and so on.

Clearly, LFS must have some means of tracking segment states and inode positions. Sprite-LFS did so by introducing a *segment usage table* which contains, among other information, number of live blocks and inodes left in segments

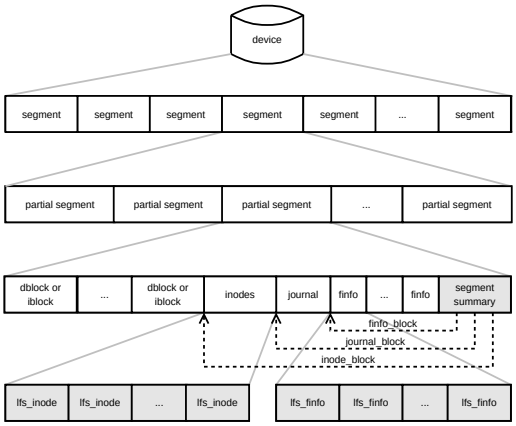


Figure 1: The internal structure of segments and partial segments.

and an *inode table* to store the inode positions. Both were fixed size stand-alone kernel structures written to the log at file system checkpoints. BSD-LFS and our implementation place both of them into an immutable regular file, visible in the file system, called the *ifile*. This approach simplifies the design because the ifile can be handled with less special-case code. Moreover, it does not impose any limit on the number of inodes in the system as the ifile can grow just like any other file.

2.1 Segment Building

All data and metadata except superblocks are written into the log consisting of segments. Therefore, the primary role of the code performing writes is to create the segments and is usually referred to as the segment building subsystem. Our implementation of LFS does not actively seek any data to write, it simply services write requests issued by the memory manager, the user or the garbage collector. In all these cases, the write requests are demands to flush a dirty page, a set of dirty pages or an inode to the disk. The basic algorithm turns out to be fairly simple:

1. Obtain a free segment from the segment management subsystem.
2. Any request to write a data block or an indirect block which fits into the current segment is carried out in the following steps: First, we schedule all necessary journal records to be written when the current partial segment is finished (see Sect. 2.3). Second, we create a new file info structure that describes the given block and plan to write it when the partial segment is about to be closed. Third, we immediately write the block itself to the end of the log. Finally, we update the segment usage table if needed. Even though we queue some metadata, the actual blocks are not queued or copied in any way. This property of the segment building subsystem is very helpful in out-of-memory situations because the allocated structures are incomparably smaller than the

written block which is about to become clean and thus reclaimable in the page cache.

3. Requests to write inodes are handled in a simpler way. First, the required journal records are also scheduled like in the previous case. The inode is then copied to an extra chunk of memory and is ready to be written before the system moves onto the next partial segment. The segment usage table may also need to be updated.

4. The subsystem checks whether there is enough room for both the blocks and all the planned metadata in the current segment. If any request cannot be safely accommodated in it, the queued inodes, journals, and file info structures are flushed to the disk together with a newly created segment summary record. Finally, we write new information about the current segment to the segment usage table and store new positions of all inodes in this segment in the inode table. The segment has been finished and the subsystem starts again from point 1.

5. A similar course of events takes place when the current partial segment must be closed because of operations like `sync`. The difference is that there is usually a substantial amount of unused space left in the current physical segment. In that case, we start a new partial segment within it.

Sprite-LFS stored the file information records and the segment summary at the end of every partial segment. It required the underlying block device layer and the disk controller to preserve the order of write requests and thus a presence of a segment summary guaranteed the whole partial segment has been written successfully. BSD-LFS did not make such assumptions, used checksums to verify a segment was valid and the authors have therefore decided to put the two structures at the beginning of a partial segment. Even though we also use checksums to verify segment integrity, the algorithm described above dictates that metadata are placed after the data and indirect blocks because the blocks are flushed to the disk before we even know how much metadata there will be, let alone what its structure will be.

We create consistent checkpoints by flushing all dirty data from the page and inode caches and by writing a consistent ifile. Unfortunately, the algorithm described above cannot produce a consistent ifile because the segment usage table is stored within the ifile and any update of it marks a new block dirty. Therefore, we first create consistent ifile partial segments in memory and flush them to disk after reaching a stable state. This is not a problem though, because creating a checkpoint is never a part of memory reclaiming and so we can safely perform substantial allocations.

2.2 Metadata Caching

In order to work effectively, any file system must buffer all data it may access repeatedly within a short period of time. The page cache is the key mechanism to buffer file data in Linux. Because virtually all file systems use it despite storing data in many very different ways, it offers a great level of control over how the data are read and written. On the other hand, traditional Linux block device based file

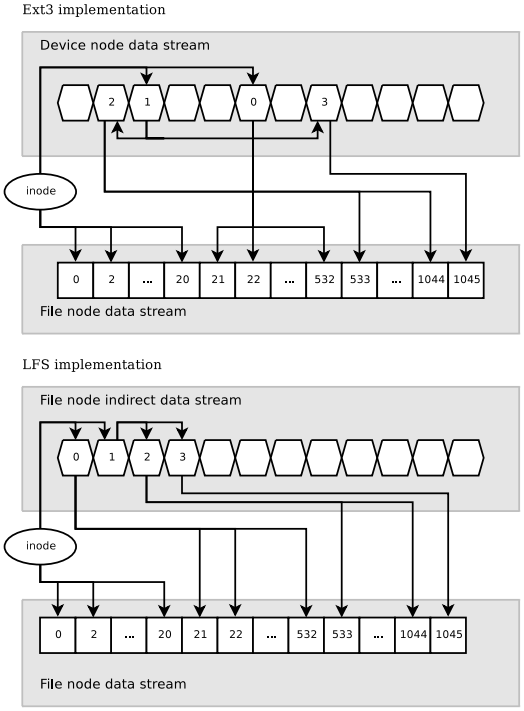


Figure 2: Indirect blocks as a special stream.

systems use the device node page cache to buffer metadata. That is convenient as long as the metadata stay in the same place on the disk because the device node cache can flush it to the spot where it was read from at any time.

Obviously, such behavior is highly undesirable in LFS. Because directories can use the page cache, new techniques to buffer indirect blocks and inodes had to be investigated. Let us consider the indirect blocks first. NILFS solved a similar problem by introducing a cache of their own. On the contrary, we wanted to unify the code that deals with direct and indirect blocks as much as possible, maximize the utilization of functionality already offered by the Linux kernel and minimize the number of special cases. We have observed that the page cache itself meets our requirements if we can create an extra stream of pages[2] per every file (Fig. 2). This solution proved to work very well. In addition, the number of places where different code paths are taken according to the type of the processed block is very small and all of them are only a few lines at most.

Inodes are compulsorily buffered by the inode cache in the Linux kernel and thus there was no question of using something else. On the other hand, the semantics of the inode cache does not perfectly suit LFS either. When the cache decides to free a dirty inode, it asks the corresponding file system to synchronously write it to the disk and discards the memory structure immediately afterwards. Neverthe-

[2]In Linux kernel terminology, this is called a mapping.

less, the segment building algorithm described in Sect. 2.1 does not and cannot actually write the inode until the current partial segment is closed. Closing it whenever an inode is written synchronously because of memory reclaiming would bring about unbearable performance overhead. This means that after the inode cache structure is discarded and before finishing the segment, the current version of the inode is present neither in the cache nor on the disk and the address in the inode table is wrong.

In order to avoid this problem, all inodes we plan to write are also added to an *ihash* hash table and kept there until the disk controller signals they were successfully written. If the kernel requests an inode in the critical time window described above, we obtain it from the ihash rather than from the disk. Moreover, the ihash also helps us to avoid writing a single inode several times into one partial segment and unnecessary inode garbage collection.

2.3 Crash Recovery

LFS currently offers two ways of recovering from an unexpected crash. The simpler one is to continue from the last checkpoint, guaranteed to be in an entirely consistent state, and discard any subsequently written data. Moreover, we also provide a roll forward utility to recover as much information as possible even if it was written after a checkpoint. This utility starts with the last checkpoint and follows the chain of segments that have been written since then as long as their checksums are correct. All entities in each of these segments are identified by examining the segment summary and file information records and the appropriate metadata are updated. If a data block is read, the utility modifies the relevant indirect block or inode. If an inode is encountered, it updates the inode table so that it points to this copy of the inode. In both cases, segment usage table must also be modified.

The utility must also deal with consistency between directories and inodes. If a crash occurs while only a part of a directory operation has been written to the disk, the link counter of an inode might not match the number of directory entries or such an entry may refer to a non-existent or even a wrong inode. The basic problem is that most directory operations affect multiple inodes and either all changes or none at all must be recovered during roll forward. BSD-LFS fights with the problem by marking all partial segments that contain an unfinished directory operation and not performing roll-forward of them unless they are followed by a segment that completes the operations. Sprite-LFS and our implementation insert a record for each directory operation to the log . These records together form a *directory operation journal*. Both file systems guarantee that the corresponding journal record appears in the log before any affected inode or directory block. Even though this approach makes roll-forward more difficult to implement, it allows recovering more data and imposes very few requirements on the implementation of the directory operations which thus can be simpler and quicker.

2.4 Directories

The log-based organization performs very well in situations where small files are manipulated intensively. Because such workloads update directories frequently, the directory manipulation operations should be performed in logarithmic time. Therefore, for directories larger than one block, we use the *htree* indexing method proposed by Danies Phillips[8] and currently used in ext3. Our implementation follows the paper closely, except that our data structures are simpler because we do not need to provide backward compatibility with early ext2 directories.

3. FREE SPACE MANAGEMENT AND GARBAGE COLLECTION

Free space management in LFS has two main goals. It provides free segments to the segment building subsystem and it recognizes and deals with "out of free space" situations. These situations must be detected by the system call handler because we cannot signal the error to the user space afterwards. Therefore, whenever we allocate a new block or an inode, we appropriately decrement a global free space counter stored in the superblock, unless it would become smaller than a certain threshold value. In that case we return an error to the user space. Conversely, whenever the user deletes an entity, we increment the same counter. The threshold value is set to 15% of the total disk size and is required to store metadata and provide some extra space for garbage collection.

We have already stressed that all live data must be copied out of a segment before reusing it. However, there may be almost no free segments available even though there is enough free space on the disk. Empty segments are created during *garbage collection* by rewriting live data from underutilized segments. There are four important issues when doing so:

1. the file system must be able to detect situations when it is essential or profitable to start cleaning,
2. the best segments to empty must be identified,
3. live data within those segments must be identified, read and appended to the current end of the log, resulting in a smaller number of near-full segments,
4. the selected segments must be reclaimed once it is safe.

We implemented the second step in the user space and the rest of the garbage collector in the kernel. All four steps are nontrivial and we will cover them in the rest of this section.

3.1 Segment Preallocation

The segment building code cannot start garbage collection at the moment it requests a new segment – it is already too late at that point. It could have been called because memory is low and so any memory allocations can either fail or block until the issued write finishes. Since identifying and reading live data from a segment may require a lot of memory allocations, both could cause a deadlock. Moreover, since the decision which segments are to be cleaned is done in the user space, any access to memory can cause a page fault and a blocking memory allocation which would also lead to a deadlock.

To avoid this situation, we track the number of dirty blocks and inodes in the cache. We compare it to the space in the

currently free segments each time a system call or the page fault handler is about to mark another block dirty. The key idea is to ensure that all dirty blocks can be written to the currently free segments at any time. This method is called *segment preallocation*. Moreover, we always reserve a number of free segments for the garbage collector and the ifile. Therefore, when a block is about to become dirty but there are not enough free segments to satisfy the constraints described in this section, we suspend the current process until enough segments are emptied and activate the garbage collector with an *emergency* message.

One exception to this rule is the ifile because its blocks are regularly marked dirty by the segment building code which cannot wait for garbage collection. Still, we must guarantee a place in free segments for dirty ifile blocks too, otherwise the system might deadlock. Therefore we add the size of an ifile to the mandatory segment reserve.

3.2 Segment Selection and Cleaning

Sprite-LFS had all components of the garbage collector incorporated in the operating system kernel. BSD-LFS has moved it to the user space so that different cleaning strategies tailored to different workloads could be easily implemented. Additionally, their garbage collector used the ifile to learn about segment utilization. Our implementation also makes the selection decisions in the user space for the same reasons. Nevertheless, we returned the rest of the cleaner back to the kernel so that it can efficiently communicate and synchronize with the segment building code through the page cache. The user space and kernel parts communicate by the *NETLINK* protocol so that there is only one communication channel independent of the on-disk format. The user space selection utility listens on the NETLINK socket for information about segment usage changes and cleaning commands. Cleaning commands are issued by the kernel when there is an imminent shortage of segments or the file system has been idle for a given period of time. Even though the kernel can support a number of segment selection utilities, so far we have implemented only one based on the cost-benefit algorithm [9]. When the utility selects a particular segment to clean it sends back a request to the kernel over the same NETLINK interface. We clean the selected segment in the kernel on the behalf and within the context of the selection task.

We process the segment by reading the relevant segment summaries, file info structures, and inodes and identifying live entities like Sprite-LFS and BSD-LFS do. When we find a live block, we read it to the page cache, mark it dirty, and pass the whole inode to the segment building code which writes all dirty pages of that inode to the end of the log. It is important to note that both read and write operations need to lock the corresponding page in the page cache. That means no process can have any page cache locked if blocked during preallocation and waiting for the garbage collector to free more segments. However, these processes originally intended to update the contents of a page and thus must release the appropriate page lock before blocking. Fortunately, the generic write functions in Linux kernel 2.6.17 and later can handle this situation and reacquire the lock if necessary. Flushing all dirty pages and not just those read by the garbage collector is deliberate because it stores adjacent

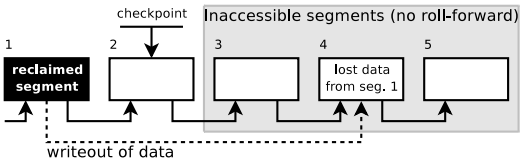


Figure 3: Roll-forward is off: Data from segment 1 were successfully moved. Segment 1 was reclaimed before checkpoint was updated resulting in loss of that data.

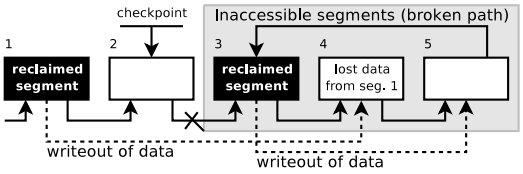


Figure 4: Roll-forward is on: Data from segment 1 and 3 were successfully moved. Segment 1 was reclaimed but this is no problem because data can be reconstructed during the roll-forward. Then segment 3 is reclaimed resulting in loss of data from segment 1 because the path of segments for roll-forward is broken.

blocks in the file next to each other on the disk. Preallocation guarantees there are free segments for both the blocks that were garbage collected and those modified by the user so this can never exhaust the segment reserve for cleaning.

3.3 Segment Reclaiming

When all data blocks, indirect blocks, and inodes in a segment have either been deleted by the user or moved to a different place on the disk, the segment becomes empty. But there are important reasons why it cannot be reused immediately. Consider the following situation: the user has modified a data block and thus a write request has been issued to store it to a new location on the disk. At the same time, the segment usage is decremented and drops to zero. If the segment was immediately reclaimed and overwritten, the block subsystem could reorder the writes so that the old copy of the block is overwritten before the new one safely lands on the disk. If a system crash occurs within this time interval the block would be lost. The segment management therefore never reuses a segment unless all segments to which data could be moved have been safely written to disk. Moreover, when roll-forward is turned off, segment reuse must be postponed until the next `sync`. Otherwise, we could not recover from crashes by simply continuing from the last checkpoint on because parts of the consistent checkpoint file system state might be overwritten (Fig. 3).

On the other hand, if roll-forward is enabled, the file system must take care not to reuse any segment that is younger than the current checkpoint. Segments form a singly-linked list which would obviously be broken by overwriting any of its items (Fig. 4).

3.4 Snapshot Implementation

When a snapshot is mounted, the real file system creates a checkpoint. We record the ifile inode position within this checkpoint and use it in the same functions that perform reading of the live data. In this way, the user has access to all data at the time of the mount as long as no part has been overwritten. To enforce this, the garbage collector does not consider segments that are older than the snapshot checkpoint and never reclaims them until the snapshot is unmounted. We currently support only one snapshot at a time but the scheme can be easily extended to support multiple concurrent ones.

When a snapshot is mounted, the free space management of the live file system behaves somehow non-intuitively. Naturally, every block which has been modified after the snapshot was mounted must be stored twice, thus occupying twice as much space on the disk. The disk free space therefore decreases even by modification of already existing data, not only when new are created. Most intriguingly, the user may run out of free space by simple file deletion because there is no space left to store the modified copy of the directory.

As stated above, over the lifetime of the snapshot, we do not reuse segments that were not free during mounting. That means the life system has only the free segments at its disposal during this time. Therefore, when the snapshot is mounted, we set the new free space to 85% of the free segments size. Conversely, when the snapshot is unmounted later on, we mark all segments which have meanwhile become empty as immediately usable and return the free space to its usual value.

4. EVALUATION

We have carried out a number of measurements[3] to evaluate the performance of our LFS implementation under different workloads and compare it to the most common Linux file system today, namely ext3. The reference computer was AMD Athlon XP 2500+ with 256MB RAM and two SATA disks. The measured file systems always resided on a 80GB Seagate Barracuda drive while the rest of the system was placed on a 120GB WD. The memory available was intentionally fairly low so that data sets bigger than the available cache did not have to be huge.

4.1 IOZone Benchmark

IOZone[1] is a filesystem benchmark capable of measuring a wide variety of file system operations. We have used it to run four series of measurements. Two of them were single-threaded and two involved eight threads. Additionally, two included the time taken by an `fsync` after writing, unlike the other two. In each series, different record and file sizes were used. The maximum file size was always 512 megabytes which was twice the amount of available RAM.

LFS performed extremely well in creation of new small files in both runs without a subsequent `fsync` (Fig. 5). This was expected because ext3 needed to read and process metadata from the disk while LFS did all processing in memory. The speed of creating files of the same size as the amount of RAM or bigger is comparable to ext3 because both file systems need to evict data from the page cache and ext3 also writes it almost sequentially because the data is new.

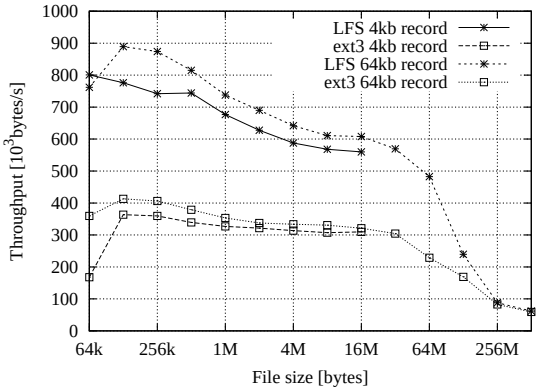


Figure 5: Single-threaded new file creation

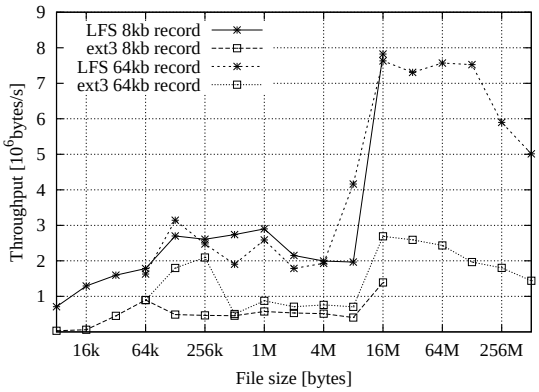


Figure 6: Multi-threaded random write to files

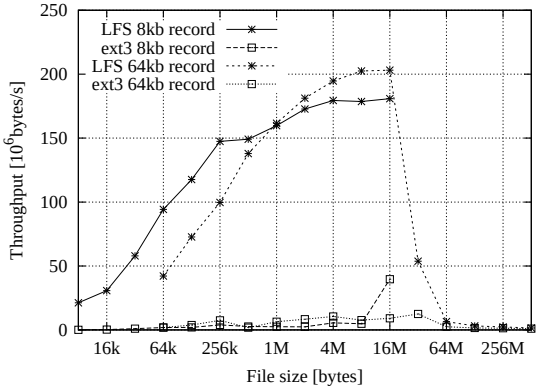


Figure 7: Multi-threaded mixed workload with fsync

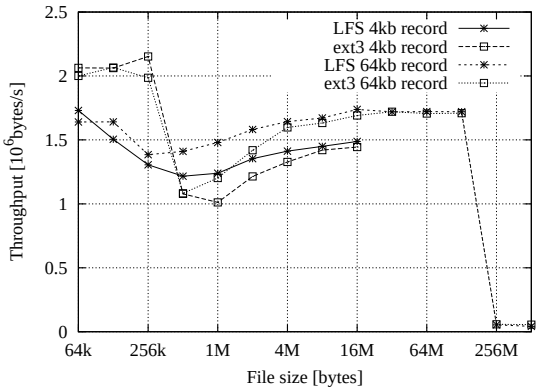


Figure 8: Single-threaded file read

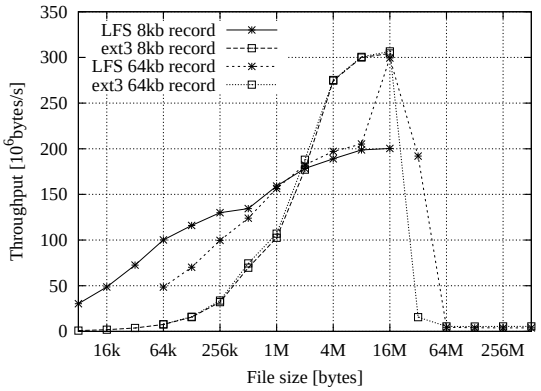


Figure 9: Multi-threaded file read

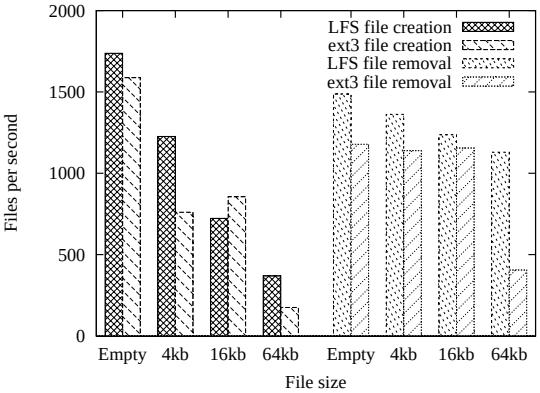


Figure 10: File creation and removal performance

On the other hand, frequent `fsync` operations mean a lot of expensive partial segment finishing and so when they were included in measurements, both file systems have performed comparably. Both file systems also produced similar results when rewriting existing files in all runs except the single-threaded one with `fsync` in which LFS was not as fast in writing small files.

LFS was better in experiments where records were overwritten randomly and immediately flushed to disk by an `fsync`, especially in the multi-threaded case (Fig. 6). LFS has also clearly outperformed ext3 in a *mixed workload with fsync* test in which each thread runs either a read or a write test on a round robin basis (Fig. 7).

Single-threaded read performance of LFS was comparable to ext3 for all but the smallest files (Fig. 8). We believe this is because ext3 implementation is more optimized rather than because of the different disk layout. Quite surprisingly, the opposite is true in the multi-threaded case where LFS is better at reading small files but worse at mid-sized ones9.

4.2 File creation and removal

LFS are known to perform very well in metadata dominated workloads [9, 11] such as creating and deleting files. In order to examine the performance of our implementation in this field, we have measured the time required by LFS and ext3 to create and delete half a million files of various sizes, including a final `sync`. The results presented in Fig. 10 show that LFS performs better, often significantly, at creating small files, except for those having 16 kilobytes. We believe ext3 performs unexpectedly well in this particular case due to such factors as cache alignment. LFS is also substantially better at deleting files, particularly mid-sized and large ones. This experiment also proved that our directory operations implementation is efficient.

4.3 Postmark Benchmark

Postmark[4] is a widely used benchmark to assess system performance under small file and generally metadata intensive workloads. It works by creating a number of small files and subsequently modifying them in so called transactions. Each transaction consists of a pair of create-or-delete and read-or-append operations. We evaluated LFS using five million postmark transactions on ten thousand files and default values of other configuration options. Postmark reported LFS was more than seven times as quick as ext3 in all measured operations. This correlates with the good results obtained in the mixed workload of the IOZone benchmark.

4.4 Garbage Collector Overhead

All measurements described in this section so far involved very little or no garbage collecting at all. However, the need to reclaim underutilized segments is a major drawback of the LFS concept. In order to asses the effect of garbage collection on write performance, we have carried out the following set of experiments. We filled a quarter, a half, and three quarters of a 10 gigabyte partition with data and then measured how much time it takes to randomly rewrite it with 8 gigabytes of data. Rewritten records have 64 kilobytes and were rewritten either with the same probability or with so called 10/90 access pattern in which 10% of hot

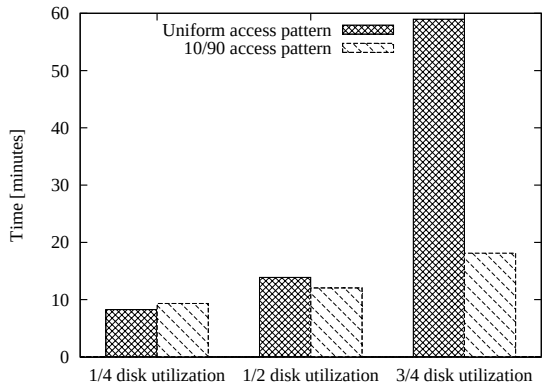


Figure 11: Garbage collector overhead

records are overwritten with 90% probability and 90% of cold records are modified with the probability of 10%. The results can be found in Fig. 11. When the 10/90 access pattern was used, the time required to complete the test when three quarters of the disk were utilized was nearly double of that when only a quarter was. On the other hand, when all records were selected with the same probability, it was over seven times as big. LFS can therefore deliver the performance presented in the previous measurements only if a substantial part of the disk is left unused. Nevertheless, the required amount depends heavily on the access pattern of a particular workload and is quite reasonable when hot data take only a small portion of disk. Moreover, the file system also carries out garbage collection in periods of inactivity which reduces the overhead when high performance is required.

5. CONCLUSION

We have introduced a novel and complete implementation of LFS for the Linux kernel with an associated user space garbage collector.

The obtained results show that our implementation outperforms ext3 in workloads with prevailing writes, especially when modifying a lot of metadata, but also when writing ordinary data randomly. Moreover, the speed of read operations is generally comparable to ext3. This performance may degrade because of garbage collection overhead, but we have shown this negative effect is small when the working set itself is small compared to the free space available.

Thus, LFS is suitable for systems such as news or mail servers. Moreover, as disks are continuously getting bigger and their seek times do not improve as much, the advantages of this layout will probably increase while the need for extra space is likely to present an ever smaller problem in the future. Finally, because our implementation is open source, other programmers and researchers may extend on the idea of LFS.

Acknowledgments

The authors are grateful to Luboš Bulej for his valuable comments.

6. REFERENCES

[1] Iozone filesystem benchmark. http://www.iozone.org, Oct 2006.

[2] T. Hrubý, M. Jambor, J. Tauš, and K. Krchák. Log-structured file system for Linux 2.6 official documentation. http://aiya.ms.mff.cuni.cz/lfs/doc/lfs.pdf, Sep 2006.

[3] T. Hrubý, M. Jambor, J. Tauš, and K. Krchák. Log-structured file system web site. http://aiya.ms.mff.cuni.cz/lfs/, Nov 2006.

[4] J. Katcher. PostMark: A new file system benchmark. Technical Report TR3022, Network Appliance, 1997.

[5] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai. The Linux implementation of a log-structured file system. *SIGOPS Oper. Syst. Rev.*, 40(3):102–107, 2006.

[6] Namesys. File system benchmarks. http://www.namesys.com/benchmarks.html.

[7] J. K. Ousterhout and F. Douglis. Beating the I/O bottleneck: A case for log-structured file systems. *Operating Systems Review*, 23(1):11–28, 1989.

[8] D. R. Phillips. A directory index for ext2. In *Proceedings of the Ottawa Linux Symposium*, pages 425–439, 2002.

[9] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 1–15, New York, NY, USA, 1991. ACM Press.

[10] M. I. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An implementation of a log-structured file system for UNIX. In *USENIX Winter*, pages 307–326, 1993.

[11] M. I. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. N. Padmanabhan. File system logging versus clustering: A performance comparison. In *USENIX Winter*, pages 249–264, 1995.

[12] A. S. Tanenbaum and A. S. Woodhull. *Operating systems (2nd ed.): Design and implementation*. Prentice-Hall, Inc., 1997.

[13] S. Tweedie. Journaling the Linux ext2fs filesystem. In *LinuxExpo '98*, 1998.

[14] wikipedia.org. Log-structured file system. http://en.wikipedia.org/wiki/Log-structured_File_System, Oct 2006.

Chapter 11

Conclusion and Future Work

We have proposed several novel approaches to mitigating the state explosion problem. In this Chapter, we briefly evaluate the proposed techniques "settled" in time, with respect to further applications and research opportunities.

Behavior protocols were enhanced with exceptions (presented in Chapter 5, alternatively presented in [7]) on demand from practice of creating formal specification of an already existing large project. However, since following projects focuses on the bottom-up development (for example [15]), the exceptions were not used. That leads to omitting the exceptions in the Kofron's proposed sequel of Behavior protocols [18], partially also due to potentially more complex implementation of the conversion to Promela. Nonetheless, we still believe the exceptions are commonly used in components and should be appropriately reflected in the specification.

The method of formal manipulation with a specification presented in Chapter 6 is very promising. Majority of the specifications (and thus models) preserves some structural properties (most method calls are synchronous, for example) which could be employed in order to reduce the size of the generated state space. Since Behavior protocols are a specification language with a relatively simple syntax, most of the reduction rules are relative easy to test and therefore the method works very well. Enhancing the rules to Extended Behavior Protocols [18] is a challenge due to parameter passing which could make the relations among components difficult to track statically. Nevertheless, reducing EPB by symbolic manipulations is a future research challenge for us. Moreover, the result could be applicable and extendable to even more complex languages such as Promela. As an aside, many reduction rules are blocked by dependences which are not present in the final model. Unfortunately, these "virtual" dependences are difficult to eliminate precisely, since a thorough analysis is as time consuming as the verification itself. If we could find a reasonable estimate or a heuristic to refine the dependences, the effi-

ciency of the reduction would be pushed further. We see a potential in the algorithm [6] based on heuristic reachability analysis.

Probably the most appealing results are presented in Chapter 8; The streamed method of distributed verification combines features hardly achievable by traditional partitioning methods (such as optimal load balancing and employing external mass storage devices). Since we have focused in safety analysis only, our future intention is to investigate whether the method is applicable also in checking liveness property. Furthermore, we would like to identify in detail the models and languages the method is suited best: As discussed, we expect higher network bandwidth than in the traditional approach and, thus, large state representations would (intuitively) lead to network saturations and verification efficiency degradation.

Although the partial-order reduction presented in Chapter 9 does not bring a new "killer" result, a very interesting finding is the possibility of notable reductions in abstract languages where asynchronous processes are so tightly synchronously coupled as in Behavior protocols. Our experience is that the partial-order reduction in the SPIN model checker produces very poor results in a one-to-one conversion from Behavior protocols to Promela.

LFS presented in Chapter 10 is an approach to providing a background for fault-tolerant streamed distributed model verification. In this case, the recovery is achieved by the operating system as a specialized data organization. Obviously, an implementation in the kernel of an operating system is very challenging and for the purpose of the verification itself very demanding. Thus, practically from the checker implementation point of view, it is beneficial to either use an already existing logging facility embedded in the operating system (as is the case of LFS) or implement an in-house solution in the scope of the model checker.

Author's Publication

[1] V. Holub, "Identifying representatives for interfering automata," in 1^{st} *International Conference on Digital Communications and Computer Applications*. Jordan University of Science and Technology, pp. 1178–1185.

[2] V. Holub and P. Tuma, "Streaming state space: A method of distributed model verification," in *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*. IEEE Computer Society, 2007, pp. 356–368.

[3] V. Holub and F. Plasil, "Reducing component systems' behavior specification," in *XXVI International Conference of the Chilean Computer Science Society*. IEEE Computer Society, 2007, pp. 356–368.

[4] F. Plasil and V. Holub, "Exceptions in component interaction protocols - a necessity," in *Dagstuhl Seminar 04511 - Architecting Systems with Trustworthy Components*, ser. Lecture Notes in Computer Science, R. Reussner, J. A. Stafford, and C. A. Szyperski, Eds., vol. 3938. Springer-Verlag, Jan 2006, pp. 227–244.

[5] V. Holub, "On distributed verification of generalized interaction models of software components," Jul 2006, presented on 20^{th} European Conference on Object-Oriented Programming (ECOOP 2006) Doctoral Symposium. [Online]. Available: www.ecoop.org/phdoos/ecoop2006ds/ds/holub.pdf

[6] ——, "State dependances in interfering automata," in *WDS'06 Proceedings of Contributed Papers: Part I - Mathematics and Computer Sciences*, J. Safrankova, Ed. Matfyzpress, Jun 2006, pp. 141–145.

[7] ——, "Enhancing behavior protocols with exceptions," in *WDS'05 Proceedings of Contributed Papers: Part I - Mathematics and Computer Sciences*, J. Safrankova, Ed. Matfyzpress, Jun 2005, pp. 30–35.

[8] M. Jambor, T. Hruby, J. Taus, K. Krchak, and V. Holub, "Implementation of a linux log-structured file system with a garbage collector," *Operating Systems Review*, vol. 41, no. 1, pp. 24–32, Jan 2007.

Other References

[9] F. Plasil, S. Visnovsky, and M. Besta, "Bounding component behavior via protocols," in *TOOLS*, vol. 30. USA: IEEE Computer Society, Aug 1999, pp. 387–398.

[10] F. Plasil and S. Visnovsky, "Behavior protocols for software components," in *IEEE Transactions on Software Engineering*, vol. 28, no. 11. IEEE, Nov 2002, pp. 1056–1076.

[11] F. Plasil, D. Balek, and R. Janecek, "SOFA/DCUP: Architecture for component trading and dynamic updating," in *International Conference on Configurable Distributed Systems (CDS)*. Washington, DC, USA: IEEE Computer Society, 1998.

[12] J. Adamek, T. Bures, P. Jezek, J. Kofron, V. Mencl, P. Parizek, and F. Plasil, "Component reliability extensions for Fractal component model." [Online]. Available: http://kraken.cs.cas.cz/ft/public/public_index.phtml

[13] J. Adamek and F. Plasil, "Component composition errors and update atomicity: Static analysis," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 17(5), pp. 363–377, Sep 2005.

[14] ——, "Erroneous architecture is a relative concept," in *Software Engineering and Applications (SEA) conference*. Cambridge, MA, USA: ACTA Press, Nov 2004, pp. 715–720.

[15] P. Jezek, J. Kofron, and F. Plasil, "Model checking of component behavior specification: A real life experience," in *International Workshop on Formal Aspects of Component Software (FACS'05)*, vol. 160. Elsevier B.V, Aug 2006, pp. 197–210.

[16] M. Mach, F. Plasil, and J. Kofron, "Behavior protocol verification: Fighting state explosion," in *International Journal of Computer and Information Science*, vol. 6. ACIS, Mar 2005, pp. 22–30.

[17] D. Balek and F. Plasil, "Software connectors and their role in component deployment," in *DAIS'01*. Kluwer, Sep 2001.

[18] J. Kofron, "Behavior protocols extensions," Ph.D. dissertation, Charles University, 2007.

[19] ——, "Enhancing behavior protocols with atomic actions," Department of software engineering, Charles University, Tech. Rep. 2005/8, 2005.

[20] B. Zimmerova, L. Brim, I. Cerna, and P. Varekova, "Component-interaction automata as a verification-oriented component-based system specification," *SIGSOFT Software Engineering Notes*, vol. 31, no. 2, Sep 2006.

[21] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, "Specifying distributed software architectures," in *5^{th} European Software Engineering Conference (ESEC 95)*, W. Schafer and P. Botella, Eds., vol. 989. London, UK: Springer-Verlag, 1995, pp. 137–153.

[22] J. Magee, J. Kramer, and D. Giannakopoulou, "Behaviour analysis of software architectures," in *TC2 First Working IFIP Conference on Software Architecture (WICSA)*. Deventer, The Netherlands: Kluwer, B.V., 1999, pp. 35–50.

[23] G. J. Holzmann, "The model checker SPIN," *Transactions on Software Engineering (TSE)*, vol. 23, no. 5, pp. 279–295, Jan 1997.

[24] S. Jabbar and S. Edelkamp, "Parallel external directed model checking with linear I/O." in *Verification, Model Checking, and Abstract Interpretation*, ser. Lecture Notes in Computer Science, E. A. Emerson and K. S. Namjoshi, Eds., vol. 3855. Springer, Jan 2006, pp. 237–251.

[25] G. Ciardo, J. Gluckman, and D. Nicol, "Distributed state space generation of discrete-state stochastic models," *INFORMS Journal on Computing*, vol. 10, no. 1, pp. 82–93, 1998.

[26] D. M. Nicol and G. Ciardo, "Automated parallelization of discrete state-space generation," *Journal of Parallel and Distributed Computing*, vol. 47, no. 2, pp. 153–167, Dec 1997.

[27] J. H. Reif, "Depth-first search is inherently sequential," in *Information Processing Letters*, vol. 20, 1985, pp. 229–234.

[28] S. Edelkamp, S. Jabbar, and S. Schrodl, "External A*," in *KI 2004: Advances in Artificial Intelligence, 27th Annual German Conference on AI*, ser. Lecture Notes in Computer Science, S. Biundo, T. W. Fruhwirth, and G. Palm, Eds., vol. 3238. Springer, 2004, pp. 226–240.

[29] S. Jabbar and S. Edelkamp, "I/O efficient directed model checking," in *Verification, Model Checking, and Abstract Interpretation (VMCAI05)*, ser. Lecture Notes in Computer Science, R. Cousot, Ed., vol. 3385. Springer-Verlag, 2005, pp. 313–329.

[30] F. Lerda and R. Sisto, "Distributed-memory model checking with SPIN," in *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*. London, UK: Springer-Verlag, 1999, pp. 22–39.

[31] U. Stern and D. L. Dill, "Parallelizing the Murφ verifier," in *9th International Conference on Computer-Aided Verification (CAV)*. Springer-Verlag, 1997, pp. 256–278.

[32] ——, "Using magnatic disk instead of main memory in the Murφ verifier," in *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*. London, UK: Springer-Verlag, 1998, pp. 172–183.

[33] D. L. Dill, "The Murφ verification system," in *CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification*. London, UK: Springer-Verlag, 1996, pp. 390–393.

[34] M. D. Jones and J. Sorber, "Parallel search for LTL violations," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 7, no. 1, pp. 31–42, 2005.

[35] M. Weiser, "Program slices: Formal, psychological, and pratical investigations of an automatic program abstraction method," Ph.D. dissertation, University of Michigan, 1979, ann Arbor.

[36] ——, "Program slicing," *IEEE Transactions on Software Engineering*, vol. 10, no. 4, pp. 352–357, 1984.

[37] ——, "Programmers use slices when debugging," *Communications in the ACM*, vol. 25, no. 7, pp. 446–452, 1982.

[38] J. R. Larus and S. Chandra, "Using tracing and dynamic slicing to tune compilers," University of Wisconsin-Madison, Tech. Rep. CS-TR-1993-1174, 1993.

[39] E. Bruneton, T. Coupaye, and J.-B. Stefani, "Recursive and dynamic software composition with sharing," in *7th International Workshop on Component-Oriented Programming (WCOP02)*, Jun 2002.

[40] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.-B. Stefani, "The Fractal component model and its support in Java," *Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems*, vol. 32, no. 11–12, pp. 1257–1284, 2002.

[41] E. Bruneton, T. Coupaye, and J.-B. Stefani, "Fractal specification." [Online]. Available: http://fractal.objectweb.org/specification/

[42] T. Kalibera and P. Tuma, "Distributed component system based on architecture description: The SOFA experience," in *On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002*. London, UK: Springer-Verlag, 2002, pp. 981–994.

[43] "Julia framework." [Online]. Available: http://fractal.objectweb.org/julia/index.html

[44] "The SOFA project." [Online]. Available: http://sofa.objectweb.org/

[45] T. Bures, P. Hnetynka, and F. Plasil, "SOFA 2.0: Balancing advanced features in a hierarchical component model," in *SERA 2006*. IEEE Computer Society, Aug 2006, pp. 40–48.

[46] P. Hnetynka, F. Plasil, T. Bures, V. Mencl, and L. Kapova, "SOFA 2.0 metamodel," Charles university, Tech. Rep. 2005/11, Dec 2005.

[47] "Speedo." [Online]. Available: http://speedo.objectweb.org/

[48] P. Hnetynka and F. Plasil, "Dynamic reconfiguration and access to services in hierarchical component models," in *CBSE 2006*, ser. Lecture Notes in Computer Science, vol. 4063. Springer-Verlag, Jun 2006, pp. 352–359.

[49] "Promela language reference." [Online]. Available: http://www.spinroot.com/spin/Man/promela.html

[50] T. Heyman, D. Geist, O. Grumberg, and A. Schuster, "Achieving scalability in parallel reachability analysis of very large circuits," in *Computer-Aided Verification, 12th International Conference*, ser. Lecture Notes in Computer Science, O. Grumberg, Ed., vol. 1855. Springer-Verlag, Jun 2000, pp. 20–35.

[51] G. Behrmann, T. Hune, and F. W. Vaandrager, "Distributing timed model checking — how the search order matters," in *CAV '00: 12th International Conference on Computer Aided Verification*. London, UK: Springer-Verlag, 2000, pp. 216–231.

[52] G. Behrmann, "A performance study of distributed timed automata reachability analysis," ser. Electronic Notes in Theoretical Computer Science, vol. 68, no. 4, Oct 2002.

[53] S. W. Ng, "Improving disk performance via latency reduction," *IEEE Transactions on Computers*, vol. 40, no. 1, pp. 22–30, Jan 1991.

[54] R. Milner, "Synthesis of communicating behaviour," in 7^{th} *Symposium on Mathematical Foundations of Computer Science*, ser. Lecture Notes in Computer Science, J. Winkowski, Ed., vol. 64. Springer, 1978, pp. 71–83.

[55] M. Hennessy and R. Milner, "On observing nondeterminism and concurrency," in *Proceedings of the 7th Colloquium on Automata, Languages and Programming*. London, UK: Springer-Verlag, 1980, pp. 299–309.

[56] R. Milner, *A Calculus of Communicating Systems*. Secaucus, NJ, USA: Springer-Verlag, 1982.

[57] C. A. R. Hoare, "A model for communicating sequential processes," in *On the Construction of Programs*, R. M. McKeag and A. M. Macnaughten, Eds. Cambridge University Press, 1980, pp. 229–254.

[58] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Communications of the ACM*, vol. 18, no. 8, pp. 453–457, 1975.

[59] C. A. R. Hoare, "Communicating sequential processes," *Communications in the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[60] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe, "A theory of Communicating sequential processes," *Journal of the ACM*, vol. 31, no. 3, pp. 560–599, 1984.

[61] J. A. Bergstra and J. W. Klop, "A convergence theorem in process algebra," in *Ten Years of Concurrency Semantics: Selected Papers of the Amsterdam Concurrency Group*, W. Bakker and J. J. M. M. Rutten, Eds. World Scientific, 1992.

[62] ——, "Process algebra for synchronous communication," *Information and Control*, vol. 60, no. 1-3, pp. 109–137, 1984.

[63] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[64] F. Tip, "A survey of program slicing techniques," *Journal of programming languages*, vol. 3, pp. 121–189, 1995.

[65] H. Agrawal, R. A. DeMillo, and E. H. Spafford, "Debugging with dynamic slicing and backtracking," *Software - Practice and Experience*, vol. 23, no. 6, pp. 589–616, 1993.

[66] K. B. Gallagher and J. R. Lyle, "Using program slicing in software maintenance," *IEEE Trans. Softw. Eng.*, vol. 17, no. 8, pp. 751–761, 1991.

[67] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, 1998.

[68] R. Milner, *Communication and Concurrency*, C. Hoare, Ed. Prentice Hall, 1989.

[69] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, parts I and II," *Information and Computation*, vol. 100, no. 1, pp. 1–40, 1992.

[70] F. Baude, D. Caromel, and M. Morel, "From distributed objects to hierarchical grid components," in *International Symposium on Distributed Objects and Applications (DOA)*, ser. Lecture Notes in Computer Science, D. C. Schmidt, R. Meersman, and Z. Tari, Eds., vol. 2888. Springer-Verlag, 2003, pp. 1226–1242.

[71] D. Conan, C. Taconet, D. Ayed, L. Chateigner, N. Kouici, and G. Bernard, "A Pro-Active middleware platform for mobile environments," in *IASTED International Conference on Software Engineering*, M. H. Hamza, Ed. ACTA Press, 2004, pp. 701–706.

[72] R. Gupta, M. J. Harrold, and M. L. Soffa, "An approach to regression testing using slicing," in *Proceedings of the International Conference on Software Maintenance 1992*, 1992, pp. 299–308.

[73] M. Harman and S. Danicic, "Using program slicing to simplify testing," *Software Testing, Verification & Reliability*, vol. 5, no. 3, pp. 143–162, 1995.

[74] J. Rilling and T. Klemola, "Identifying comprehension bottlenecks using program slicing and cognitive complexity metric," in *11th International Workshop on Program Comprehension (IWPC 2003)*. IEEE Computer Society, 2003, pp. 115–124.

[75] M. Harman, D. Binkley, and S. Danicic, "Amorphous program slicing," *Journal of Systems and Software*, vol. 68, no. 1, pp. 45–64, 2003.

[76] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi, "On the temporal analysis of fairness," in *POPL '80: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM Press, 1980, pp. 163–173.

[77] P. Brada, "Component change and version identification in SOFA," in *SOFSEM '99: Proceedings of the 26th Conference on Current Trends in Theory and Practice of Informatics on Theory and Practice of Informatics*. London, UK: Springer-Verlag, 1999, pp. 360–368.

[78] ——, "Metadata support for safe component upgrades," in *COMPSAC '02: Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 1017–1021.

[79] T. Kalibera, L. Bulej, and P. Tuma, "Automated detection of performance regressions: The Mono experience," in *MASCOTS '05: Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 183–190.

[80] L. Bulej and T. Bures, "Eliminating execution overhead of disabled optional features in connectors," in *Software Architecture, Third European Workshop, EWSA*, ser. Lecture Notes In Computer Science, V. Gruhn and F. Oquendo, Eds., vol. 4344. Springer, 2006, pp. 50–65.

[81] ——, "Using connectors for deployment of heterogeneous applications in the context of OMG D&C specification," in *1st International Conference on Interoperability of Enterprise Software and Applications (INTEROP-ESA 2005)*, 2005, pp. 349–360.

[82] R. H. Reussner, H. W. Schmidt, and I. H. Poernomo, "Reliability prediction for component-based software architectures," *Journal of Systems and Software*, vol. 66, no. 3, pp. 241–252, 2003.

[83] S. Becker, H. Koziolek, and R. Reussner, "Model-based performance prediction with the Palladio component model," in *WOSP '07: Proceedings of the 6th international workshop on Software and performance*. New York, NY, USA: ACM Press, 2007, pp. 54–65.

[84] J. Happe, H. Koziolek, and R. Reussner, "Parametric performance contracts for software components with concurrent behaviour," *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 182, pp. 91–106, 2007.

[85] H. Koziolek and J. Happe, "A quality of service driven development process model for component-based software systems," in *9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE2006)*, ser. Lecture Notes In Computer Science. Springer-Verlag, Jul 2006.

[86] S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni, "Model-based performance prediction in software development: A survey," *IEEE Transactions on Software Engineering*, vol. 30, no. 5, pp. 295–310, 2004.

[87] "Modelling contest: Common component modelling example (CoCoME)." [Online]. Available: http://agrausch.informatik.uni-kl.de/CoCoME

Appendix A

Full-fledged Example of the Reduction Presented in Chapter 6

In this appendix, we elaborate the example from the Chapter 6 in detail. The whole protocol depicted in Fig. A.1 is reduced in nine steps. Each step is dedicated to exactly one reduction rule which is applied multiple times. We discuss the conditions which must be satisfied to guarantee the correctness of the reduction. Note that the order of reduction rules has been chosen to fit well to our description and does not correspond to the order suggested in Chapter 6.

Step 1: Reduction of external events First, we reduce external events by the rule B4. External events are (method call/acceptance abbreviated):

- `?IArbitratorLifetimeController.Start`,
- `?ILogin.GetTokenIdFromIpAddress`,
- `?ILogin.LoginWithFlyTicketId`,
- `!IFlyTicketAuth.CreateToken`,
- `?ILogin.LoginWithFrequentFlyerId`,
- `?ILogin.LoginWithAccountId`,
- `!IAccountAuth.CreateToken`,
- `?ILogin.Logout`,
- `?IDhcpCallback.IpAddressInvalidated_1`,
- `?IAccount.GenerateRandomAccountId`,
- `?IAccount.CreateAccount`, and

Arbitrator

```
( ?IArbitratorLifetimeController.Start^ ;
  !ITokenLifetimeController.Start^ ;
  [?ITokenLifetimeController.Start$,
    !IArbitratorLifetimeController.Start$] );
(
  (
    ?ILogin.GetTokenIdFromIpAddress +
    ?ILogin.LoginWithFlyTicketId {
      !IFlyTicketAuth.CreateToken ;
      (!IFirewall.DisablePortBlock + NULL) }
    +
    ?ILogin.LoginWithFrequentFlyerId {
      !IFreqFlyerAuth.CreateToken ;
      (!IFirewall.DisablePortBlock + NULL) }
    +
    ?ILogin.LoginWithAccountId {
      !IAccountAuth.CreateToken ;
      (!IFirewall.DisablePortBlock + NULL) }
    +
    ?ILogin.Logout {
      !IToken.InvalidateAndSave_1 }
  )* |
  ?ITokenCallback.TokenInvalidated_1 {
    !IFirewall.EnablePortBlock_1 }*
  |
  ?ITokenCallback.TokenInvalidated_2 {
    !IFirewall.EnablePortBlock_2 }*
  |
  ?ITokenCallback.TokenInvalidated_3 {
    !IFirewall.EnablePortBlock_3 }*
  |
  ?IDhcpCallback.IpAddressInvalidated_1 {
    !IToken.InvalidateAndSave_2 }*
)
```

Token

```
?ITokenLifetimeController.Start ;
( ?IToken.InvalidateAndSave_1 {
    (!IAccount.AjustAccountPrepaidTime_1 + NULL);
    !ITokenCallback.TokenInvalidated_1 }* |
  ?IToken.InvalidateAndSave_2 {
    (!IAccount.AjustAccountPrepaidTime_2 + NULL);
    !ITokenCallback.TokenInvalidated_2 }* |
  (
    (!IAccount.AjustAccountPrepaidTime_3 + NULL);
    !ITokenCallback.TokenInvalidated_3 )*
)
```

AccountDatabase

```
( (
    ?IAccount.GenerateRandomAccountId +
    ?IAccount.CreateAccount +
    ?IAccount.RechargeAccount {
      !ICardCenter.Withdraw }
  )* |
  ?IAccount.AjustAccountPrepaidTime_1* |
  ?IAccount.AjustAccountPrepaidTime_2* |
  ?IAccount.AjustAccountPrepaidTime_3*
)
```

CardCenter

```
( ?ICardCenter.Withdraw* )
```

Firewall

```
( ?IFirewall.EnablePortBlock_1* |
  ?IFirewall.EnablePortBlock_2* |
  ?IFirewall.EnablePortBlock_3* |
  ?IFirewall.DisablePortBlock*
)
```

Figure A.1: Airport Internet providing service [15, 3] (Fig. 2.5)

- `?IAccount.RechargeAccount`.

We can reduce these transitions under the conditions of ν-elimination, particullary missing accepting transitions from the end of the reduced transition. Immediately, we can reduce:

- `?IArbitratorLifetimeController.Start`$\uparrow$,
- `?ILogin.GetTokenIdFromIpAddress`$\uparrow$,
- `?ILogin.LoginWithFlyTicketId`$\uparrow$,
- `?ILogin.LoginWithFrequentFlyerId`$\uparrow$,
- `?ILogin.LoginWithAccountId`$\uparrow$,
- `?ILogin.Logout`$\uparrow$,
- `?IDhcpCallback.IpAddressInvalidated_1`$\uparrow$,
- `?IAccount.GenerateRandomAccountId`$\uparrow$,
- `?IAccount.CreateAccount`$\uparrow$, and
- `?IAccount.RechargeAccount`$\uparrow$,

but rest of the transitions are blocked by sequel accepting transitions. However, most of them are actually events from the list above. Thus, after reducing external events from the first list, we are able to reduce:

- `!IArbitratorLifetimeController.Start`$\downarrow$,
- `!ILogin.GetTokenIdFromIpAddress`$\downarrow$,
- `!ILogin.LoginWithFlyTicketId`$\downarrow$,
- `?IFlyTicketAuth.CreateToken`$\downarrow$,
- `!ILogin.LoginWithFrequentFlyerId`$\downarrow$,
- `?IAccountAuth.CreateToken`$\downarrow$,
- `!ILogin.LoginWithAccountId`$\downarrow$,
- `!ILogin.Logout`$\downarrow$,
- `!IDhcpCallback.IpAddressInvalidated_1`$\downarrow$,
- `!IAccount.GenerateRandomAccountId`$\downarrow$,
- `!IAccount.CreateAccount`$\downarrow$, and
- `!IAccount.RechargeAccount`$\downarrow$.

Now, method calls `!IFlyTicketAuth.CreateToken`$\uparrow$ (two instances) and `!IAccountAuth.CreateToken`$\uparrow$ remains. Fortunately, since these were blocked by `?IFlyTicketAuth.CreateToken`$\downarrow$ and `?IAccountAuth.CreateToken`$\uparrow$ and have already been removed in the previous reduction, we can reduce them as well. Finally the specification takes the form of Fig. A.2.

Arbitrator

```
!ITokenLifetimeController.Start ;
(
  (
    (!IFirewall.DisablePortBlock + null)
    +
    (!IFirewall.DisablePortBlock + null)
    +
    (!IFirewall.DisablePortBlock + null)
    +
    !IToken.InvalidateAndSave_1
  )*
  |
  ?ITokenCallback.TokenInvalidated_1 {
    !IFirewall.EnablePortBlock_1
  }*
  |
  ?ITokenCallback.TokenInvalidated_2 {
    !IFirewall.EnablePortBlock_2
  }*
  |
  ?ITokenCallback.TokenInvalidated_3 {
    !IFirewall.EnablePortBlock_3
  }*
  |
  !IToken.InvalidateAndSave_2*
)
```

Token

```
?ITokenLifetimeController.Start
;
(
  ?IToken.InvalidateAndSave_1 {
    (!IAccount.AjustAccountPrepaidTime_1
      + null);
    !ITokenCallback.TokenInvalidated_1
  }*
  |
  ?IToken.InvalidateAndSave_2 {
    (!IAccount.AjustAccountPrepaidTime_2
      + null);
    !ITokenCallback.TokenInvalidated_2
  }*
  |
  (
    (!IAccount.AjustAccountPrepaidTime_3
      + null);
    !ITokenCallback.TokenInvalidated_3
  )*
)
```

AccountDatabase

```
(
  !ICardCenter.Withdraw*
  |
  ?IAccount.AjustAccountPrepaidTime_1*
  |
  ?IAccount.AjustAccountPrepaidTime_2*
  |
  ?IAccount.AjustAccountPrepaidTime_3*
)
```

CardCenter

```
(
  ?ICardCenter.Withdraw*
)
```

Firewall

```
(
  ?IFirewall.EnablePortBlock_1*
  |
  ?IFirewall.EnablePortBlock_2*
  |
  ?IFirewall.EnablePortBlock_3*
  |
  ?IFirewall.DisablePortBlock*
)
```

Figure A.2: Step 1 — Reduction of external events

Step 2: Simple method call reduction In this step, we reduce simple method calls (and acceptances) of

- `ITokenLifetimeController.Start`,
- `IFirewall.DisablePortBlock`,
- `IFirewall.EnablePortBlock_{1,2,3}`,
- `IAccount.AjustAccountPrepaidTime_{1,2,3}`, and
- `ICardCenter.Withdraw`.

To preserve the correctness, we have to test all the conditions of the ν-elimination on the second part of method invocation and acceptance:

- `!ITokenLifetimeController.Start` is at the beginning of `Arbitrator`. We want to reduce `?ITokenLifetimeController.Start`$\downarrow$ which has outgoing accepting transitions `?ITokenCallback.TokenInvalidated_{1,2,3}`$\uparrow$ and potentially can violate ν-elimination conditions. But the counterparts of these transitions lay in `Token` and are sequentially blocked by `?ITokenLifetimeController.Start`$\uparrow$. Thus they cannot cause a bad activity error at the start of `?ITokenLifetimeController.Start`$\downarrow$ and the correctness of the ν-elimination is preserved.
- A similar argument holds for `!ITokenLifetimeController.Start`$\downarrow$. Here, the potentialy dangerous actions are `?IToken.InvalidateAndSave_{1,2,3}`$\uparrow$. However, those are sequentially blocked at the Arbitrator by `!ITokenLifetimeController.Start`$\uparrow$.
- `!IFirewall.DisablePortBlock`$\downarrow$ is in three instances in Arbitrator. Since all the transitions we can continue to are emitting, the conditions for ν-closure holds.
- The same argument used in the previous bullet holds for `!Firewall.EnablePortBlock_{1,2,3}`$\downarrow$, `!IAccount.AjustAccountPrepaidTime_{1,2,3}`$\downarrow$, `!ICardCenter.Withdraw`$\downarrow$, `?IAccount.AjustAccountPrepaidTime_{1,2,3}`$\downarrow$, and `?IFirewall.DisablePortBlock`$\downarrow$.

After the reductions, the protocol takes the form of Fig. A.3.

Step 3: Initial state transitions This step consists of a reduction `!ITokenLifetimeController.Start`$\uparrow$ and `?ITokenLifetimeController.Start`$\uparrow$ according to the reduction rule B5. The reduction is allowed since `Arbitrator` and `Token` cannot continue without proceeding τ`ITokenLifetimeController.Start`$\uparrow$, `Firewall` awaits blocked `Arbitrator`, and `AccountDatabase` can only communicate with `CardCenter` and vice versa (Fig. A.4).

Step 4: Redundant state reduction After the reduction in step 3 (Fig. A.4), `Arbitrator`, `Token`, `AccountDatabase`, and `Firewall` consist of several independent parts connected together via a single state for each component. Since these parts behave concurrently, the connecting states are redundant and can be reduced according to the reducing rule C3.

The effect of the reduction is depicted in Fig. A.5. Four components are split up to 16 parts.

Arbitrator

```
!ITokenLifetimeController.Start^ ;
(
  (
    (!IFirewall.DisablePortBlock^ + null)
    +
    (!IFirewall.DisablePortBlock^ + null)
    +
    (!IFirewall.DisablePortBlock^ + null)
    +
    !IToken.InvalidateAndSave_1
  )*
  |
  ?ITokenCallback.TokenInvalidated_1 {
    !IFirewall.EnablePortBlock_1^
  }*
  |
  ?ITokenCallback.TokenInvalidated_2 {
    !IFirewall.EnablePortBlock_2^
  }*
  |
  ?ITokenCallback.TokenInvalidated_3 {
    !IFirewall.EnablePortBlock_3^
  }*
  |
  !IToken.InvalidateAndSave_2*
)
```

Token

```
?ITokenLifetimeController.Start^
;
(
  ?IToken.InvalidateAndSave_1 {
    (!IAccount.AjustAccountPrepaidTime_1^
      + null);
    !ITokenCallback.TokenInvalidated_1
  }*
  |
  ?IToken.InvalidateAndSave_2 {
    (!IAccount.AjustAccountPrepaidTime_2^
      + null);
    !ITokenCallback.TokenInvalidated_2
  }*
  |
  (
    (!IAccount.AjustAccountPrepaidTime_3^
      + null);
    !ITokenCallback.TokenInvalidated_3
  )*
)
```

AccountDatabase

```
(
  !ICardCenter.Withdraw^*
  |
  ?IAccount.AjustAccountPrepaidTime_1^*
  |
  ?IAccount.AjustAccountPrepaidTime_2^*
  |
  ?IAccount.AjustAccountPrepaidTime_3^*
)
```

CardCenter

```
(
  ?ICardCenter.Withdraw^*
)
```

Firewall

```
(
  ?IFirewall.EnablePortBlock_1^*
  |
  ?IFirewall.EnablePortBlock_2^*
  |
  ?IFirewall.EnablePortBlock_3^*
  |
  ?IFirewall.DisablePortBlock^*
)
```

Figure A.3: Step 2 — Simple method call reduction

Arbitrator

```
(
  (
    (!IFirewall.DisablePortBlock^ + null)
    +
    (!IFirewall.DisablePortBlock^ + null)
    +
    (!IFirewall.DisablePortBlock^ + null)
    +
    !IToken.InvalidateAndSave_1
  )*
  |
  ?ITokenCallback.TokenInvalidated_1 {
    !IFirewall.EnablePortBlock_1^
  }*
  |
  ?ITokenCallback.TokenInvalidated_2 {
    !IFirewall.EnablePortBlock_2^
  }*
  |
  ?ITokenCallback.TokenInvalidated_3 {
    !IFirewall.EnablePortBlock_3^
  }*
  |
  !IToken.InvalidateAndSave_2*
)
```

Token

```
(
  ?IToken.InvalidateAndSave_1 {
    (!IAccount.AjustAccountPrepaidTime_1^
      + null);
    !ITokenCallback.TokenInvalidated_1
  }*
  |
  ?IToken.InvalidateAndSave_2 {
    (!IAccount.AjustAccountPrepaidTime_2^
      + null);
    !ITokenCallback.TokenInvalidated_2
  }*
  |
  (
    (!IAccount.AjustAccountPrepaidTime_3^
      + null);
    !ITokenCallback.TokenInvalidated_3
  )*
)
```

AccountDatabase

```
(
  !ICardCenter.Withdraw^*
  |
  ?IAccount.AjustAccountPrepaidTime_1^*
  |
  ?IAccount.AjustAccountPrepaidTime_2^*
  |
  ?IAccount.AjustAccountPrepaidTime_3^*
)
```

CardCenter

```
(
  ?ICardCenter.Withdraw^*
)
```

Firewall

```
(
  ?IFirewall.EnablePortBlock_1^*
  |
  ?IFirewall.EnablePortBlock_2^*
  |
  ?IFirewall.EnablePortBlock_3^*
  |
  ?IFirewall.DisablePortBlock^*
)
```

Figure A.4: Step 3 — Initial state transitions

Arbitrator

```
(
  (!IFirewall.DisablePortBlock^ + null)
  +
  (!IFirewall.DisablePortBlock^ + null)
  +
  (!IFirewall.DisablePortBlock^ + null)
  +
  !IToken.InvalidateAndSave_1
)*

?ITokenCallback.TokenInvalidated_1 {
  !IFirewall.EnablePortBlock_1^
}*

?ITokenCallback.TokenInvalidated_2 {
  !IFirewall.EnablePortBlock_2^
}*

?ITokenCallback.TokenInvalidated_3 {
  !IFirewall.EnablePortBlock_3^
}*

!IToken.InvalidateAndSave_2*
```

Token

```
?IToken.InvalidateAndSave_1 {
  (!IAccount.AjustAccountPrepaidTime_1^
    + null);
  !ITokenCallback.TokenInvalidated_1
}*

?IToken.InvalidateAndSave_2 {
  (!IAccount.AjustAccountPrepaidTime_2^
    + null);
  !ITokenCallback.TokenInvalidated_2
}*

(
  (!IAccount.AjustAccountPrepaidTime_3^
    + null);
  !ITokenCallback.TokenInvalidated_3
)*
```

AccountDatabase

```
!ICardCenter.Withdraw^*

?IAccount.AjustAccountPrepaidTime_1^*

?IAccount.AjustAccountPrepaidTime_2^*

?IAccount.AjustAccountPrepaidTime_3^*
```

CardCenter

```
(
  ?ICardCenter.Withdraw^*
)
```

Firewall

```
?IFirewall.EnablePortBlock_1^*

?IFirewall.EnablePortBlock_2^*

?IFirewall.EnablePortBlock_3^*

?IFirewall.DisablePortBlock^*
```

Figure A.5: Step 4 — Redundant state reduction

Arbitrator

```
(
  (!IFirewall.DisablePortBlock^ + null)
  +
  (!IFirewall.DisablePortBlock^ + null)
  +
  (!IFirewall.DisablePortBlock^ + null)
  +
  !IToken.InvalidateAndSave_1
)*

?ITokenCallback.TokenInvalidated_1 {
  !IFirewall.EnablePortBlock_1^
}*

?ITokenCallback.TokenInvalidated_2 {
  !IFirewall.EnablePortBlock_2^
}*

?ITokenCallback.TokenInvalidated_3 {
  !IFirewall.EnablePortBlock_3^
}*

!IToken.InvalidateAndSave_2*
```

Token

```
?IToken.InvalidateAndSave_1 {
  (!IAccount.AjustAccountPrepaidTime_1^
    + null);
  !ITokenCallback.TokenInvalidated_1
}*

?IToken.InvalidateAndSave_2 {
  (!IAccount.AjustAccountPrepaidTime_2^
    + null);
  !ITokenCallback.TokenInvalidated_2
}*

(
  (!IAccount.AjustAccountPrepaidTime_3^
    + null);
  !ITokenCallback.TokenInvalidated_3
)*
```

AccountDatabase

```
!ICardCenter.Withdraw^*

?IAccount.AjustAccountPrepaidTime_1^*

?IAccount.AjustAccountPrepaidTime_2^*

?IAccount.AjustAccountPrepaidTime_3^*
```

CardCenter

```
(
  ?ICardCenter.Withdraw^*
)
```

Firewall

```
?IFirewall.EnablePortBlock_1^*

?IFirewall.EnablePortBlock_2^*

?IFirewall.EnablePortBlock_3^*

?IFirewall.DisablePortBlock^*
```

Figure A.6: Step 5 — Simple cycle reduction

Step 5: Simple cycle reduction After the splitting of `AccountDatabase` and `Firewall` in the previous step, we can reduce eight simple cycles according to the rule B3:

- `?IAccount.AjustAccountPrepaidTime_{1,2,3}`↑
- `?ICardCenter.Withdraw`↑
- `?IFirewall.EnablePortBlock_{1,2,3}`↑, and
- `?IFirewall.DisablePortBlock`↑,

and counterparts:

- `!IFirewall.DisablePortBlock`↑
- `!IFirewall.EnablePortBlock_{1,2,3}`
- `!IAccount.AjustAccountPrepaidTime_{1,2,3}`, and
- `!ICardCenter.Withdraw`↑.

The reduction removes all these calls which results in empty protocols. Thus the resulting behavior in Fig. A.6 is empty for `AccountDatabase`, `CardCenter`, and `Firewall`.

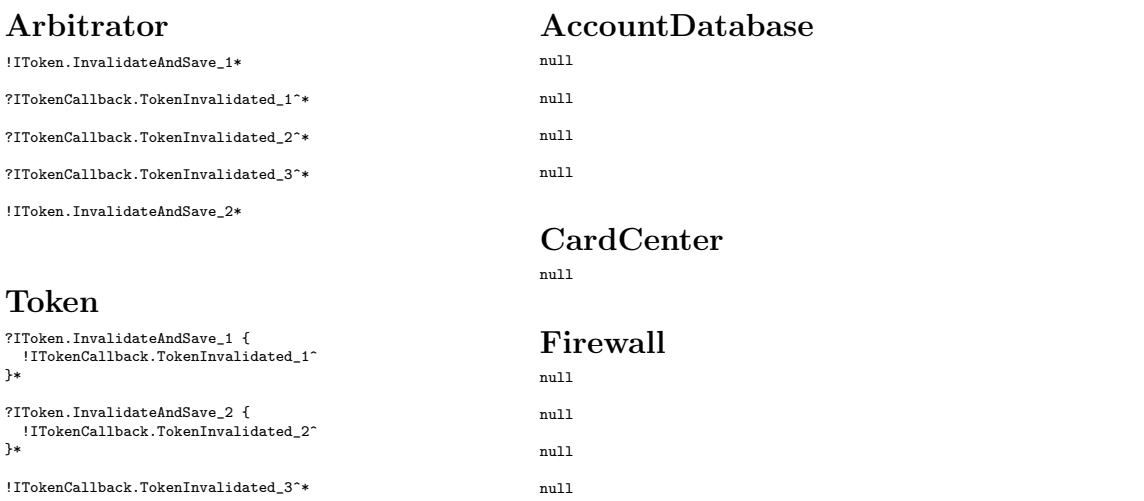


Figure A.7: Step 6 — Simple method call reduction

Step 6: Simple method call reduction In this step, we reduce simple method calls of `ITokenCallback.TokenInvalidated_{1,2,3}`↑ (Fig. A.7). Because the sequel actions of `?ITokenCallback.TokenInvalidated_{1,2}`↓ are only emit actions `!IToken.InvalidateAndSave_{1,2}`↓ and the sequel action of `?ITokenCallback.TokenInvalidated_3`↓ is only the emit action `!ITokenCallback.TokenInvalidated_3`↑, the conditions of the ν-elimination are satisfied.

Step 7: Simple cycle reduction Simple method call reductions in step 6 enables simple cycle reductions of `ITokenCallback.TokenInvalidated_{1,2,3}` (Fig. A.8).

Step 8: Simple method call reduction Method calls `IToken.InvalidateAndSave_{1,2}` can be reduced according to simple method call reduction rule B1. Since these method calls are the only behavior remaining, the conditions of the ν-elimination are satisfied. The result of the elimination is depicted in Fig. A.9.

Step 9: Simple cycle reduction Behaviors `?IToken.InvalidateAndSave_1`↑`*` and `?IToken.InvalidateAndSave_2`↑`*` are simple calls and can be eliminated according to the simple cycle reduction B3. The reduction causes `Arbitrator` and `Token` to consists of empty behavior parts (Fig. A.10).

Arbitrator

!IToken.InvalidateAndSave_1*

null

null

null

!IToken.InvalidateAndSave_2*

Token

?IToken.InvalidateAndSave_1*

?IToken.InvalidateAndSave_2*

null

AccountDatabase

null

null

null

null

CardCenter

null

Firewall

null

null

null

null

Figure A.8: Step 7 — Simple cycle reduction

Arbitrator

!IToken.InvalidateAndSave_1*

null

null

null

!IToken.InvalidateAndSave_2*

Token

?IToken.InvalidateAndSave_1*

?IToken.InvalidateAndSave_2*

null

AccountDatabase

null

null

null

null

CardCenter

null

Firewall

null

null

null

null

Figure A.9: Step 8 — Simple method call reduction

Arbitrator

null

null

null

null

null

Token

null

null

null

AccountDatabase

null

null

null

null

CardCenter

null

Firewall

null

null

null

null

Figure A.10: Step 9 — Simple cycle reduction